实验 4： Memory Management

目录

软件框图

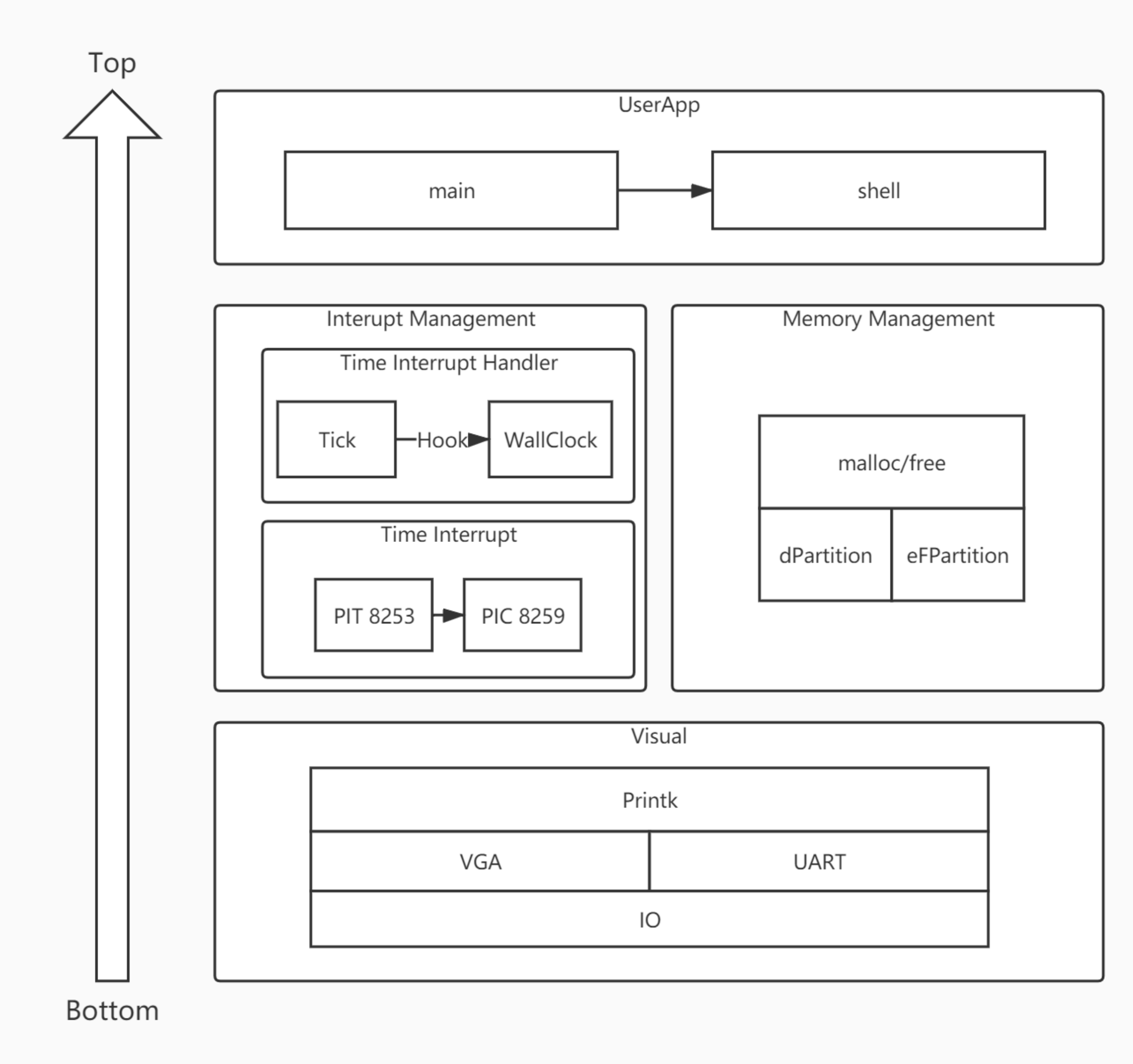
- 软件结构

  本次实验主要开发内存管理功能，通过静态等大小分区和动态分区实现malloc/free及kmalloc/kfree。

  此外，在Shell中加入了AddNewCmd，可以动态的加入新的指令，这些指令占据的空间动态分配得到的。

  最后，在接口方面整理出了userInterface.h，便于用户程序调用。

- 结构图（底层在下，顶层在上）：

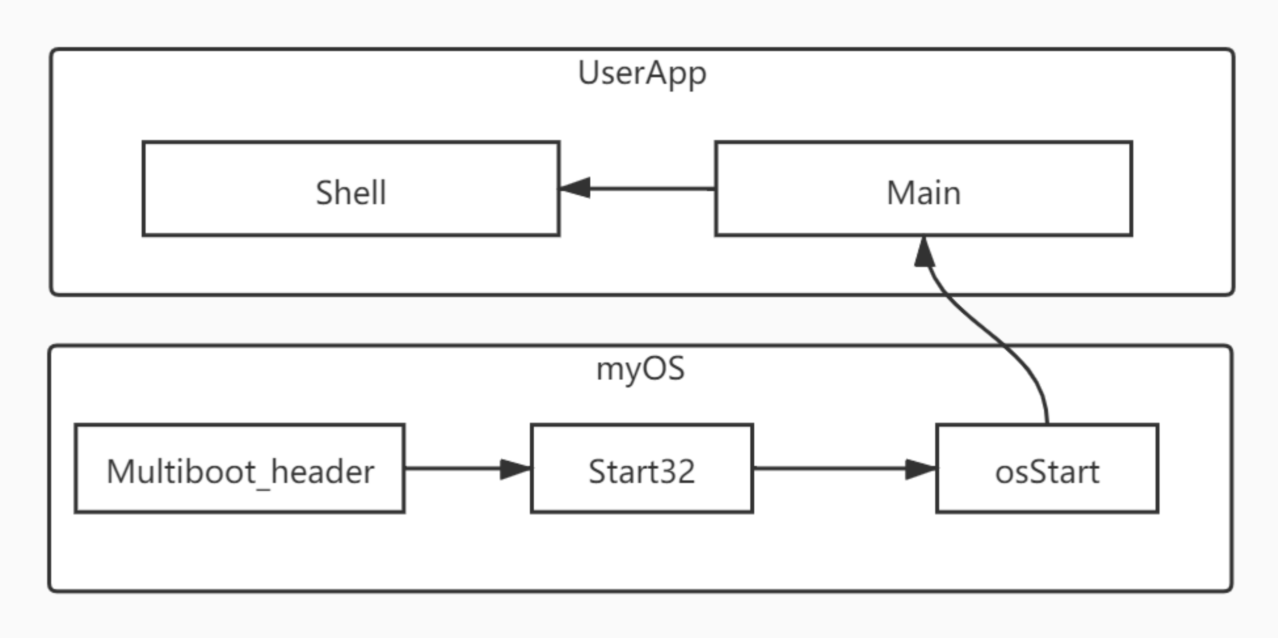


主流程

- 流程说明

主流程从Multiboot_header开始，首先进入Start32。在Start32中，程序进行了堆栈的初始化、IDT的初始化等必要的准备工作，然后将控制权移交到osStart。在osStart中，进行PIT、PIT的初始化，并开启中断，之后调用用户程序入口函数Main。Main调用Shell的开启子程序，进入控制台。

- 流程图



功能模块

概述

软件的唯一新增的模块是内存管理模块，对命令行模块添加了动态添加命令功能。

内存管理——内存检测算法

- 内存检测算法

```
void writeWord(unsigned long addr, short write){
    __asm__ __volatile__("movw %0,(%1)"::"a"(write),"b"(addr));
}
void readWord(unsigned long addr, short *read){
    __asm__ __volatile__("movw (%1),%0":"=a"(*read):"b"(addr));
}

void memTest(unsigned long start, unsigned long grainSize){

    unsigned long addr,step,tail,total;
    addr = start<0x400?0x400:start;
    tail = addr + grainSize - 2;
    step = grainSize<2?2:grainSize;
    int flag;
    short data,check,write1,write2;
    total=-1;
    write1=0x55AA;
    write2=0xAA55;


    //get memory start
    flag=0;
    while(flag!=4){
```

```
        flag=0;
        readWord(addr,&data);
        writeWord(addr,write1);
        readWord(addr,&check);
        flag += (check==write1);
        writeWord(addr,write2);
        readWord(addr,&check);
        flag += (check==write2);
        writeWord(addr,data);


        readWord(tail,&data);
        writeWord(tail,write1);
        readWord(tail,&check);
        flag += (check==write1);
        writeWord(tail,write2);
        readWord(tail,&check);
        flag += (check==write2);
        writeWord(tail,data);

        if(flag!=4) addr+=grainSize;
    }

    pMemStart = addr;

    //get memory end
    flag=4;
    while(flag==4){
        flag=0;
        readWord(addr,&data);
        writeWord(addr,write1);
        readWord(addr,&check);
        flag += (check==write1);
        writeWord(addr,write2);
        readWord(addr,&check);
        flag += (check==write2);
        writeWord(addr,data);


        readWord(tail,&data);
        writeWord(tail,write1);
        readWord(tail,&check);
        flag += (check==write1);
        writeWord(tail,write2);
        readWord(tail,&check);
        flag += (check==write2);
        writeWord(tail,data);

        if(flag==4){
            addr+=grainSize;
            pMemSize+=grainSize;
        }
    }
}
```

```
```

内存管理——静态分区

- 静态分区算法——Init 创建静态分区和每个等大小内存块的数据结构，建立内存块链表。

```
unsigned long eFPartitionInit(unsigned long start, unsigned long perSize, unsigned long n){
    //Init efpHeader
    struct eFPartition *efp = (struct eFPartition *) start;
    alignby8(&perSize);
    efp->perSize = perSize;
    efp->totalN = n;
    efp->firstFree = start + sizeof(struct eFPartition);

    //Init EEB Chain
    struct EEB * eeb;
    unsigned long addr = efp->firstFree;
    for(int i=0;i<n;i++){
        eeb = (struct EEB *) addr;
        eeb -> next_start = addr + perSize;
        addr += perSize;
    }
    eeb -> next_start = 0;
    eFPartitionWalkByAddr((unsigned long) efp);

    return (unsigned long)efp;
}
```

- 静态分区算法——Alloc 分配一个位置最靠前的内存块，成功返回内存块起始地址，失败返回0。

```
unsigned long eFPartitionAlloc(unsigned long EFPHandler){
    struct eFPartition * efp = (struct eFPartition *) EFPHandler;
    struct EEB * eeb = (struct EEB *) efp->firstFree;

    //alloc fail return 0
    if(efp->firstFree > EFPHandler+eFPartitionTotalSize(efp->perSize,efp->totalN))
return 0;

    //alloc succeed return eeb handler
    efp->firstFree = eeb->next_start;
    return (unsigned long) eeb;
}
```

- 静态分区算法——Free 释放掉某个内存块之前的所有内存块，成功返回1，失败返回0。

```
unsigned long eFPartitionFree(unsigned long EFPHandler,unsigned long mbStart){
    struct eFPartition * efp = (struct eFPartition *) EFPHandler;
```

```
    if(mbStart==0) mbStart = EFPHandler + eFPartitionTotalSize(efp->perSize,efp-
>totalN);
    efp->firstFree = EFPHandler + sizeof(struct eFPartition);
    struct EEB *eeb = (struct EEB *) efp->firstFree;
    int cnt = 0;
    //free all blocks ahead of mbStart
    while((unsigned long) eeb < mbStart){
        eeb -> next_start  = (unsigned long) eeb + efp->perSize;
        eeb = (struct EEB *) ((unsigned long) eeb + efp->perSize);
        cnt++;
    }
    eeb = (struct EEB *) ((unsigned long) eeb - efp->perSize);
    if(cnt==efp->totalN) eeb->next_start = 0;

    return 1;
}
```

内存管理——动态分区

- 动态分区——Init 创建动态分区和初始内存块的数据结构，成功返回分区起始地址，失败返回0；

```
unsigned long dPartitionInit(unsigned long start, unsigned long totalSize){
    //return start if succeed, return 0 if fail

    //totalSize should be bigger
    if(totalSize < HEADERSIZE + HEADERSIZE + 8) return 0;

    //init dPHeader
    struct dPartition *dp = (struct dPartition *) start;
    dp -> size = totalSize;
    dp -> firstFreeStart = start + HEADERSIZE;
    //Init EMB
    struct EMB *emb = (struct EMB *)(dp -> firstFreeStart);
    emb -> size = totalSize - HEADERSIZE - HEADERSIZE;
    emb -> nextStart = 0;

    //init succeed
    return start;
}
```

- 动态分区——Alloc 动态分区用FirstFit策略分配内存，成功返回分配内存起始地址，失败返回0。

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

    //illegal size
    if(size <= 0) return 0;
```

```
    struct dPartition *dPart = (struct dPartition *) dp;
    struct EMB *pre = 0;
    struct EMB *emb = (struct EMB *) dPart -> firstFreeStart;

    size += 4 ;//add at least 4-byte fence between embs
    alignby8(&size);

    while(emb){

        //allocate with current emb?
        if(emb->size >= size){
            //split current emb?
            if(emb -> size - size >= HEADERSIZE + 8){

                struct EMB *next = (struct EMB *) ((unsigned long) emb + size +
HEADERSIZE);
                next -> size = emb -> size - size - HEADERSIZE;
                next -> nextStart = emb -> nextStart;
                emb -> size = size;//update current emb

                if(pre == 0) dPart -> firstFreeStart = (unsigned long) next;
                else pre -> nextStart = (unsigned long) next;

                return (unsigned long) emb;

            }
            else{
                if(pre == 0) dPart -> firstFreeStart = (unsigned long) emb-
>nextStart;
                else pre -> nextStart = (unsigned long) emb->nextStart;

                return (unsigned long) emb;
            }
        }
        pre = emb;
        emb = (struct EMB *) emb -> nextStart;
    }

    return 0;
}
```

- 动态分区——Free 基于FirstFit策略的内存释放，维护空闲内存块链表，实现了需要链表前合并和后合并的操作。成功返回1，失败返回0。

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

    struct dPartition *dPart = (struct dPartition *) dp;
    struct EMB *emb = (struct EMB *) dPart -> firstFreeStart;
    struct EMB *tar = (struct EMB *) start;

    //illegal start
```

```
    if(start < dp + HEADERSIZE || start >= dp + dPart -> size){
        myPrintk(0xf,"here\n");
        return 0;
    }


    unsigned long beg = (unsigned long) tar;
    unsigned long end = (unsigned long) tar + HEADERSIZE + tar -> size;

    struct EMB *pre, *next;
    pre = next = 0;


    while(emb){
        if((unsigned long) emb < (unsigned long) tar) pre = emb;
        else if((unsigned long) emb > (unsigned long) tar){
            next = emb;
            break;
        }

        emb = (struct EMB *) emb -> nextStart;
    }

        if(next){
            //merge to next
            if(end == (unsigned long) next){
                tar -> nextStart = next -> nextStart;
                tar -> size += next -> size + HEADERSIZE;
            }
            else tar -> nextStart = (unsigned long) next;
        }
        else tar -> nextStart = 0;

        if(pre){
            //merge to pre
            if(beg == (unsigned long) pre + HEADERSIZE + pre -> size){
                pre -> nextStart = tar -> nextStart;
                pre -> size += tar -> size + HEADERSIZE;
            }
            else pre -> nextStart = (unsigned long) tar;
        }
        else dPart -> firstFreeStart = (unsigned long) tar;

        //free succeed
        return 1;
}
```

- 基于动态分区的malloc/free(kmallc/kfree完全相同)

```
unsigned long malloc(unsigned long size){
    //dPartition's Alloc
    if(pMemHandler) return dPartitionAlloc(pMemHandler,size);
```

```
    else return 0;
}

unsigned long free(unsigned long start){
    //dPartition's Free
    if(pMemHandler) return dPartitionFree(pMemHandler, start);
    else return 0;
}
```

命令行模块

- 命令的数据结构 同时声明了cmds这个二重指针。

```
struct command {
    char cmd[32];//maxlen = 32
    int (*func)(int argc, unsigned char **argv);
    void (*help_func)(void);
    char desc[64];//maxlen = 64
}**cmds;
```

- 动态添加命令

```
void addNewCmd( unsigned char *cmd,                             //命令名
                int (*func)(int argc, unsigned char **argv),    //命令入口
                void (*help_func)(void),                        //该命令的help入
口，可为空
                unsigned char* description)                     //该命令的描述

{

    cmds[cmdcnt] = (struct command *) malloc(sizeof(struct command));
    myPrintk(0xf,"0x%x\n",cmds[cmdcnt]);
    strcpy(cmd, cmds[cmdcnt]->cmd);
    cmds[cmdcnt]->func = func;
    cmds[cmdcnt]->help_func = help_func;
    strcpy(description, cmds[cmdcnt]->desc);
    cmdcnt++;
}
```

- 装载预置命令 调用AddNewCmd添加预置命令，实际上所有命令都是动态声明的，其所用空间都是动态分配的。

```
void initShell(){
    cmds = (struct command **) malloc(sizeof(unsigned long) * MAXCMDS);
    addNewCmd("cmd", cmd_handler, NULL, "list all commands");
    addNewCmd("help", help_handler, help_help, "help [cmd]");
```

```
    addNewCmd("cls", cls_handler, NULL, "clear screen");
    addNewCmd("exit", exit_handler, NULL, "shutdown shell");
}
```

源代码说明

- 代码组织

```
|---- lab4/
    |---- src/
        |---- source2img.sh     生成elf脚本
        |---- myOS/
            |---- userInterface.h
            |---- start32.S
            |---- osStart.c
            |---- dev/
                |---- i8253.c
                |---- i8259A.c
                |---- uart.c
                |---- vga.c
            |---- i386/
                |---- io.c
                |---- io.h
                |---- irqs.c
            |---- kernel/
                |---- mem/
                    |---- eFPartition.c
                    |---- dPartition.c
                    |---- malloc.c
                    |---- pMemInit.c
                |---- tick.c
                |---- wallClock.c
            |---- printk/
                |---- myPrintk.c
                |---- vsprintf.c
            |---- include/
                |---- i8295.h
                |---- i8259A.h
                |---- io.h
                |---- irqs.h
                |---- kmem.h
                |---- mem.h
                |---- myPrintf.h
                |---- myPrintk.h
                |---- tick.h
                |---- uart.h
                |---- vga.h
                |---- vsprintf.h
                |---- wallClock.h
        |---- userApp/
            |---- main.c
```

```
            |---- shell.c
            |---- shell.h
            |---- memTestCase.c
            |---- memTestCase.h
        |---- multibootHeader/
            |---- multibootHeader.S
```

- Makefile 组织

```
include $(SRC_RT)/myOS/Makefile
include $(SRC_RT)/userApp/Makefile
```

```
|---- lab4/
    |---- src/
        |---- myOS/
            |---- dev/
            |---- i386/
            |---- kernel/
                |---- mem
            |---- printk/
        |---- userApp/
```

地址空间说明

- ld文件

```
SECTIONS {
    . = 1M;
    .text : {
        *(.multiboot_header)
        . = ALIGN(8);
        *(.text)
    }

    . = ALIGN(16);
    .data       : { *(.data*) }

    . = ALIGN(16);
    .bss        :
    {
        __bss_start = .;
        _bss_start = .;
        *(.bss)
        __bss_end = .;
    }
    . = ALIGN(16);
    _end = .;
```

```
    . = ALIGN(512);
}
```

- 地址空间表

Offset	Field	Macro
0	.code	
1M	.text	
ALIGN(16)	.data	
ALIGN(16)	.bss	__bss_start, _bss_start
		_bss_end
ALIGN(16)		_end

编译过程说明

- 主Makefile

```
OS_OBJS       = ${MYOS_OBJS} ${USER_APP_OBJS}

output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf

output/%.o : %.S
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
```

- 说明

  根据Makefile分为两步：编译和链接。

  第一步，编译汇编代码(*.S)和c代码(*.c)并输出对象文件(*.o)。

  第二步，将这些对象文件链接并输出可执行可链接文件(myOS.elf)。

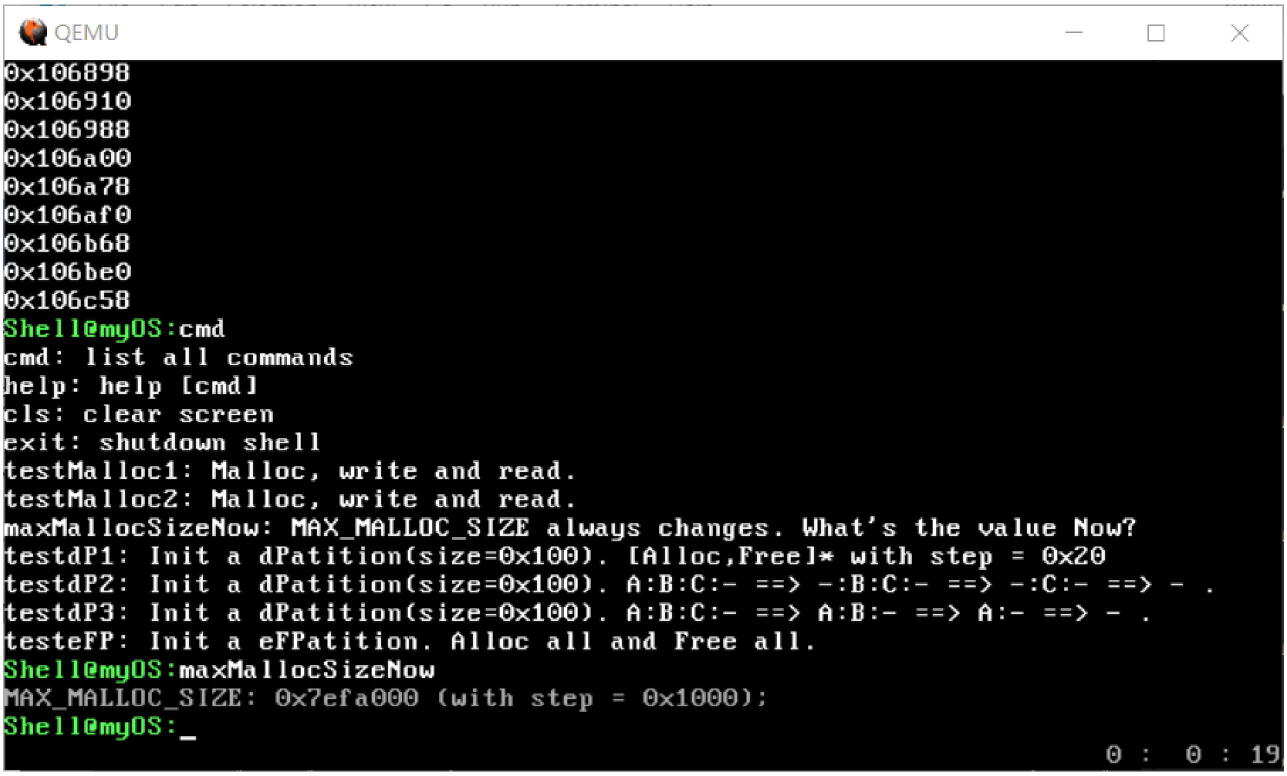
运行和运行结果说明

- 运行

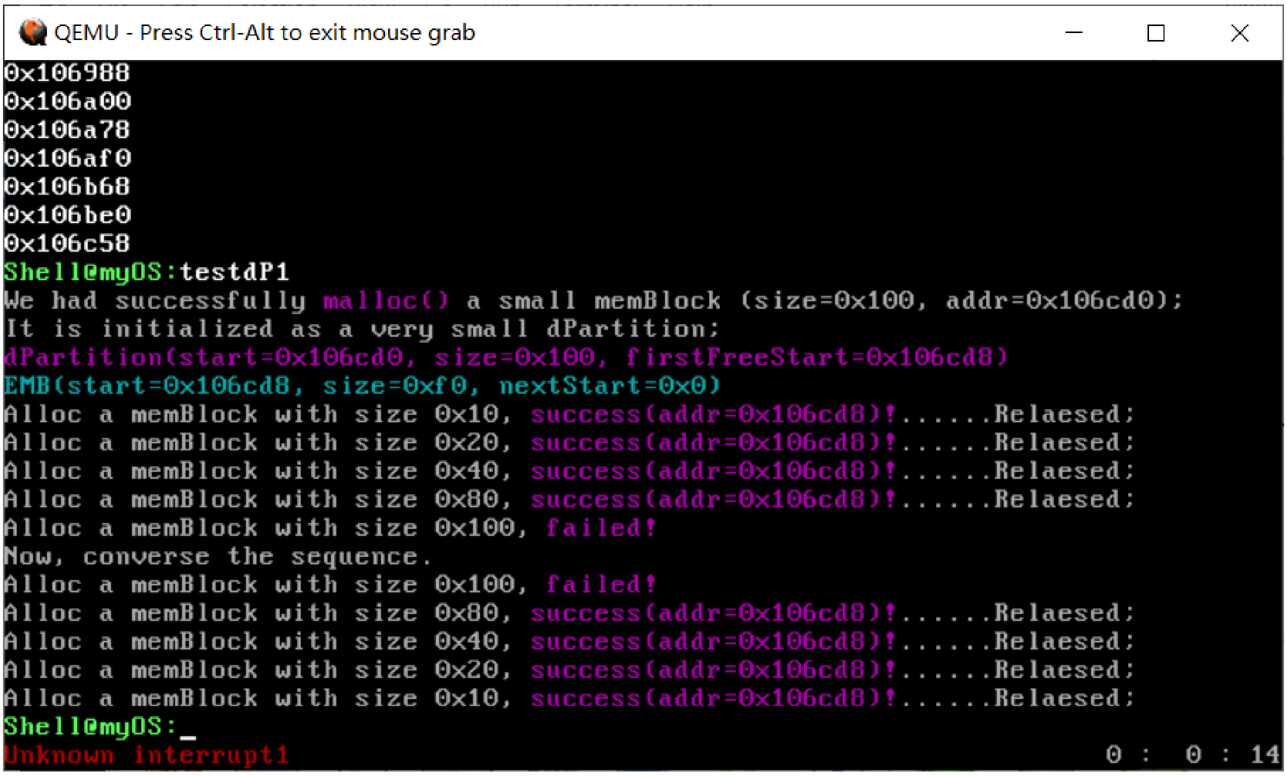
  执行命令：`qemu-system-i386 -kernel output/myOS.elf -serial pty &

  将之前编译链接生成的elf文件，加载到qemu中运行。

- 运行结果



可以看到，添加的命令都能显示出来。



各测试用例测试结果正常

遇到的问题及解决

问题：命令数据结构中的函数要怎么添加？

解决方案：命令的处理函数事实上占用的是代码段的空间，这段空间是在1M内的，不需要内存管理。只需记录函数指针即可。