OS lab4

PB21051012 刘祥辉

实验目标

掌握内存分配的基本方法，学会使用firstfit进行内存分配
加深对操作系统的理解

源代码说明

eFPartition.c:

使用链表结构进行连接，要求和助教的实验文档写的一样

```
#include "../../include/myPrintk.h"
#define TRUE  1
#define FALSE 0
// eFPartition是表示整个内存的数据结构
typedef struct eFPartition{
    unsigned long totalN;
    unsigned long perSize;  // unit: byte
    unsigned long firstFree;
}eFPartition;   // 占12个字节

#define eFPartition_size 12

void showeFPartition(struct eFPartition *efp){
    myPrintk(0x5,"eFPartition(start=0x%x, totalN=0x%x, perSize=0x%x,
firstFree=0x%x)\n", efp, efp->totalN, efp->perSize, efp->firstFree);
}

// 一个EEB表示一个空闲可用的Block
typedef struct EEB {
    unsigned long next_start;
    unsigned long is_allocated;
}EEB;   // 占8个字节

#define EEB_size 8

// void showEEB(EEB *eeb) {
//  myPrintk(0x7, "EEB (addr = 0x%x)\n", (unsigned long)eeb);
// }
// void showEEB(EEB *eeb) {
//  myPrintk(0x7, "EEB (start = 0x%x, next = 0x%x)\n", (unsigned long)eeb, eeb-
>next_start);
// }
void showEEB(EEB *eeb) {
    myPrintk(0x7, "EEB (addr = 0x%x)\n", (unsigned long)eeb, eeb->next_start);
}


void eFPartitionWalkByAddr(unsigned long efp){
    // TODO
```

```
    /*功能：本函数是为了方便查看和调试的。
    1. 打印eFPartiiton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）
    */
    showeFPartition((eFPartition*)efp);
    unsigned long p;
    for(p=(*(eFPartition*)efp).firstFree; p ; p=(*(EEB*)p).next_start){
        if( (*(EEB*)p).is_allocated==TRUE)
            showEEB((EEB*)p);
    }
}


unsigned long eFPartitionTotalSize(unsigned long perSize, unsigned long n){
    // TODO
    /*功能：计算占用空间的实际大小，并将这个结果返回
    1. 根据参数persize（每个大小）和n个数计算总大小，注意persize的对齐。
        例如persize是31字节，你想8字节对齐，那么计算大小实际代入的一个块的大小就是32字节。
    2. 同时还需要注意“隔离带”EEB的存在也会占用4字节的空间。
        typedef struct EEB {
            unsigned long next_start;
        }EEB;
    3. 最后别忘记加上eFPartition这个数据结构的大小，因为它也占一定的空间。
    */
    unsigned long alignedSize = (perSize + 3) & (~3); //四字节对齐
    return n*(sizeof(EEB)+alignedSize) + sizeof(eFPartition);
}

unsigned long eFPartitionInit(unsigned long start, unsigned long perSize,
unsigned long n){
    // TODO
    /*功能：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的大小。
    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
    注意的地方：
        1.EEB类型的数据的存在本身就占用了一定的空间。
    */
    unsigned long alignedSize = (perSize + 3) & (~3); //四字节对齐
    unsigned long i=0x0;
    eFPartition ef = {n,alignedSize,start+sizeof(eFPartition)};
    *(eFPartition*)start = ef;
    EEB eb;
    while(i<n){
        eb.is_allocated=FALSE;
        eb.next_start = start+sizeof(eFPartition)+(i+1)*
(sizeof(EEB)+alignedSize);
        *( (EEB*)(start+sizeof(eFPartition)+i*(sizeof(EEB)+alignedSize))) = eb;
        i++;
    }
    ((EEB*)(start+sizeof(eFPartition)+(n-1)*(sizeof(EEB)+alignedSize) ) )-
>next_start=0;
    return start;
}
```

```
unsigned long eFPartitionAlloc(unsigned long EFPHandler){
    // TODO
    /*功能：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
    注意的地方：
        1.EEB类型的数据的存在本身就占用了一定的空间。
    */
    unsigned long eeb = ( (eFPartition*)EFPHandler)->firstFree;
    while( eeb){
        if( ((EEB*)eeb)->is_allocated==FALSE){
            ((EEB*)eeb)->is_allocated=TRUE;
            return eeb+sizeof(EEB);
        }
        else eeb = ( (EEB*)eeb)->next_start;
    }
    return 0;
}


unsigned long eFPartitionFree(unsigned long EFPHandler,unsigned long mbStart){
    // TODO
    /*功能：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。
    */
    //myPrintk(0x7,"mbStart=%x\n",mbStart);
    //myPrintk(0x7,"str\n");
    unsigned long ps = ((eFPartition*)EFPHandler)->firstFree;
    mbStart -= sizeof(EEB);
    // int i=5;
    while(ps>0){
        // myPrintk(0x7,"i'm in loop\n");
        // i--;
        // if(i<=0) break;
        if(mbStart==ps){
            //if(((EEB*)ps)->is_allocated == TRUE)
                ((EEB*)ps)->is_allocated = FALSE;
            return 1;
        }
        else ps = ((EEB*)ps)->next_start;
    }
    return 0;
}
```

dFPartition.c:

这里我偷了个懒没有选择使用链表结构，而是在EMB快里加上了is_allocated的tag，在寻找可分配内存的时候直接查看tag是否为TRUE即可。

```
#include "../../include/myPrintk.h"

#define FALSE 0
#define TRUE  1
```

```c
//dPartition 是整个动态分区内存的数据结构
typedef struct dPartition{
    unsigned long size;
    unsigned long firstFreeStart;
} dPartition;   //共占8个字节

#define dPartition_size ((unsigned long)0x8)

void showdPartition(struct dPartition *dp){
    myPrintk(0x5,"dPartition(start=0x%x, size=0x%x, firstFreeStart=0x%x)\n", dp,
dp->size,dp->firstFreeStart);
}

// EMB 是每一个block的数据结构，userdata可以暂时不用管。
typedef struct EMB{
    unsigned long size;
    unsigned long is_allocated;
    union {
        unsigned long nextStart;    // if free: pointer to next block
         unsigned long userData;        // if allocated, belongs to user
    };
    unsigned long preStart;
} EMB;  //共占D个字节

#define EMB_size ((unsigned long)0x10)

void showEMB(struct EMB * emb){
    myPrintk(0x3,"EMB(start=0x%x, size=0x%x, nextStart=0x%x)\n", emb, emb->size,
emb->nextStart);
}

unsigned long dPartitionInit(unsigned long start, unsigned long totalSize){
    // TODO
    /*功能：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)。
    注意有两个地方的大小问题：
        第一个是由于内存肯定要有一个EMB和一个dPartition，totalSize肯定要比这两个加起来大。
        第二个注意EMB的size属性不是totalsize，因为dPartition和EMB自身都需要要占空间。
    */
    dPartition dp  = {totalSize - dPartition_size, start+dPartition_size};
    *(dPartition*)start = dp;
    EMB        emb = {totalSize - dPartition_size - EMB_size,FALSE,0,start};
    *(EMB*)(start + dPartition_size) =  emb;
    return start;
}


void dPartitionWalkByAddr(unsigned long dp){
    // TODO
    /*功能：本函数遍历输出EMB 方便调试
    1. 先打印dP的信息，可调用上面的showdPartition。
    2. 然后按地址的大小遍历EMB，对于每一个EMB，可以调用上面的showEMB输出其信息
    */
    showdPartition((dPartition*)dp);
    unsigned long p;
```

```c
    for(p=(*(dPartition*)dp).firstFreeStart; p ; p=(*(EMB*)p).nextStart){
        if((*(EMB*)p).is_allocated==TRUE)
            showEMB((EMB*)p);
    }
}


//================firstfit, order: address, low-->high=====================
/**
 * return value: addr (without overhead, can directly used by user)
**/

unsigned long dPartitionAllocFirstFit(unsigned long dp, unsigned long size){
    // TODO
    /*功能：分配一个空间
    1. 使用firstfit的算法分配空间,
    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
    注意的地方:
        1.EMB类型的数据的存在本身就占用了一定的空间。
    */
    unsigned long p = (*(dPartition*)dp).firstFreeStart;
    while(p){
        if(((EMB*)p)->is_allocated==FALSE && ((EMB*)p)->size >= (size+EMB_size))
            break;
        p = ((EMB*)p)->nextStart;
    }
    if(!p)
        return 0x0; //空间不足
    else if( (!((EMB*)p)->nextStart) &&((EMB*)p)->size>=size+EMB_size){
        //位于链表末尾
        unsigned long remained_mem_tail = (*(EMB*)p).size-size-EMB_size;
        EMB* pre = (EMB*)p;
        EMB* next;
        EMB  new_emb_tail = {remained_mem_tail, FALSE,0,p};
        next  = (EMB*)((char*)pre + size + EMB_size);
        *next = new_emb_tail;
        pre->is_allocated = TRUE;
        pre->nextStart = (unsigned long)next;
        pre->size = size;
    }
    else{
        //位于链表中间
        EMB* pre = (EMB*)p;
        EMB* next;
        next  = (EMB*)((char*)pre + size + EMB_size);
        unsigned long remained_mem_middle = pre->size - size - EMB_size;
        EMB new_emb_middle = {remained_mem_middle,FALSE,pre->nextStart,(unsigned
long)pre};
        *next = new_emb_middle;
        pre->is_allocated = TRUE;
        ((EMB*)pre->nextStart)->preStart = (unsigned long)next;
        pre->nextStart = (unsigned long)next;
        pre->size = size;
    }
    return p+EMB_size;
    //位于中间和尾端的情况部分好像可以合并？不过我懒得合并了...
```

```
}

unsigned long find_start(unsigned long dp,unsigned long start){
    unsigned long emb = ((dPartition*)dp)->firstFreeStart;
    while(emb){
        if(emb == start) return 1;
        emb = ((EMB*)emb)->nextStart;
    }
    return 0;
}

unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned long start){
    // TODO
    /*功能：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
    */
    EMB* pre;
    EMB* middle;
    EMB* next;
    start -= sizeof(EMB);
    if(!find_start(dp,start)) return 0;  //未找到start对应的EMB块，结束程序
    middle = (EMB*)start;
    middle->is_allocated = FALSE;
    pre  = (EMB*)(middle->preStart);
    next = (EMB*)(middle->nextStart);
    //前后链表均可合并
    if((unsigned long)pre!=dp&&(unsigned long)next){
        if(!pre->is_allocated&&!next->is_allocated){
            pre->size += EMB_size*2 + middle->size + next->size;
            pre->nextStart = next->nextStart;
            if( next->nextStart)
                ((EMB*)next->nextStart)->preStart = (unsigned long)pre;
            return 1;
        }
    }

    //判断与上一链表是否能合并并处理
    if((unsigned long)pre!=dp)
        if(pre->is_allocated == FALSE){
            pre->size += middle->size + EMB_size;
            pre->nextStart = middle->nextStart; //当middle为尾端的时候也成立
            if(middle->nextStart)
                ((EMB*)(middle->nextStart))->preStart = (unsigned long)pre;
        }
    //判断与下一节点能否合并
    if(middle->nextStart){
        next = (EMB*)(middle->nextStart);
        if(next->is_allocated == FALSE){
            middle->size += next->size + EMB_size;
            middle->nextStart = next->nextStart;
            if(((EMB*)next->nextStart)->nextStart)
                ( (EMB*)(((EMB*)next->nextStart)->nextStart))->preStart =
(unsigned long)middle;
        }
```

```
    }
    return 1;
}

// 进行封装，此处默认firstfit分配算法，当然也可以使用其他fit，不限制。
unsigned long dPartitionAlloc(unsigned long dp, unsigned long size){
    return dPartitionAllocFirstFit(dp,size);
}

unsigned long dPartitionFree(unsigned long   dp, unsigned long start){
    return dPartitionFreeFirstFit(dp,start);
}
```

shell.c

先使用malloc为结构体分配一段空间，然后复制相应的字符串即可。

```
void addNewCmd(unsigned char *cmd,
        int (*func)(int argc, unsigned char **argv),
        void (*help_func)(void),
        unsigned char* desc) {
    //TODO
    command *newCmd = (command *)MYmalloc(sizeof(command));
    myPrintf(0x7,"addr=%x\n",newCmd);
    char *cmd_new = (char *)MYmalloc(myStrlen(cmd) + 1);
    char *desc_new = (char *)MYmalloc(myStrlen(desc) + 1);
    if (!newCmd || !cmd_new || !desc_new)
        return;

    newCmd->cmd = cmd_new;
    newCmd->desc = desc_new;
    newCmd->func = func;
    newCmd->help_func = help_func;
    myStrcpy(newCmd->cmd, cmd);
    myStrcpy(newCmd->desc, desc);

    // insert to the head of list
    newCmd->next = ourCmds;
    ourCmds = newCmd;
}
```

问题回答

1. 请写出动态分配算法的malloc接口是如何实现的（即malloc函数调用了哪个函数，这个函数又调用了哪个函数...）

ans:由malloc.c文件中

```
unsigned long MYmalloc(unsigned long size) {
    return dPartitionAlloc(uMemHandler, size);
}
```

得知先调用了`dPartitionAlloc`函数，接着发现在dPartition.c中对`dPartitionAlloc`函数进行了封装，

```
unsigned long dPartitionAlloc(unsigned long dp, unsigned long size){
    return dPartitionAllocFirstFit(dp,size);
}
```

在本次实验中默认是用了firstfit算法，调用了`dPartitionAllocFirstFit`函数，在此函数中进行了内存分配并返回地址。

2. 运行 memTestCaseInit 那些新增的shell命令，会出现什么结果，即打印出什么信息（截图放到报告中）？是否符合你的预期，为什么会出现这样的结果。（详细地讲一两个运行结果，大同小异的可以从简

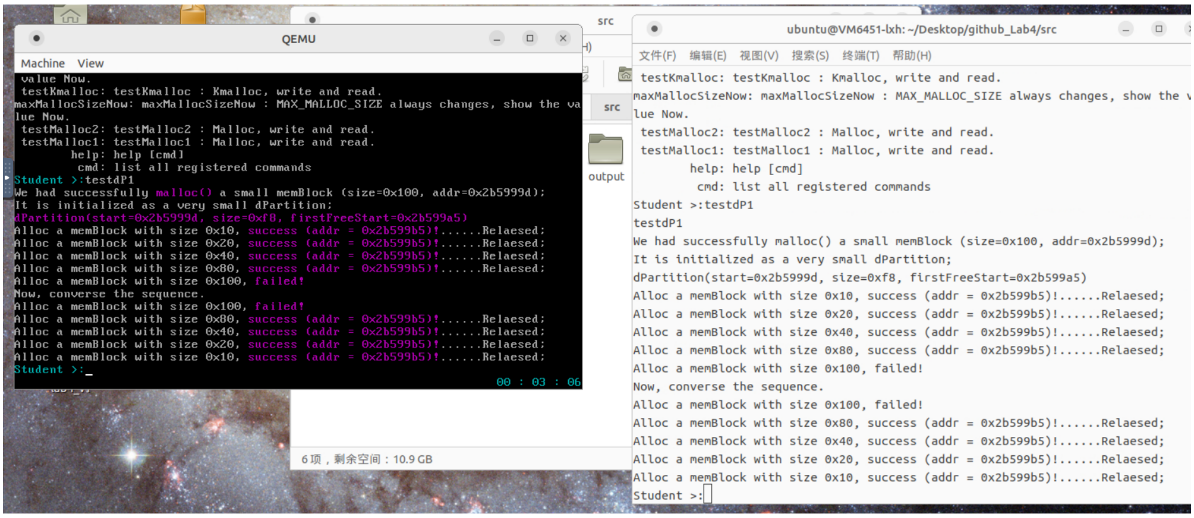
ans: memTestCseInit新增了测试shell指令，会打印出内存分配过程中的详细信息。

以testdP1为例：

```
int testdP1(int argc, unsigned char **argv){
    unsigned long x,x1,xHandler;
    int i, tsize = 0x100;
    x = MYmalloc(tsize);
    if (x){
        myPrintf(0x7, "We had successfully ");
        myPrintf(0x5, "malloc()");
        myPrintf(0x7, " a small memBlock (size=0x%x, addr=0x%x);\n", tsize,x);

        myPrintf(0x7, "It is initialized as a very small dPartition;\n");
        xHandler = dPartitionInit(x,tsize);
        dPartitionWalkByAddr(x);

        i=0x10;
        while(1){
            x1 = dPartitionAlloc(xHandler,i);
            myPrintf(0x7, "Alloc a memBlock with size 0x%x, ", i);
            if(x1) {
                myPrintf(0x5, "success (addr = 0x%x)!",x1);
                dPartitionFree(xHandler,x1);
                myPrintf(0x7, "......Relaesed;\n");
            } else {
                myPrintf(0x5, "failed!\n");
                break;
            }

            i <<= 1;
        }

        myPrintf(0x7,"Now, converse the sequence.\n");
        while(i >= 0x10){
            x1 = dPartitionAlloc(xHandler,i);
            myPrintf(0x7, "Alloc a memBlock with size 0x%x, ", i);
            if(x1) {
                myPrintf(0x5, "success (addr = 0x%x)!",x1);
                dPartitionFree(xHandler,x1);
                myPrintf(0x7, "......Relaesed;\n");
            } else myPrintf(0x5, "failed!\n");

            i >>= 1;
        }
        free(x);
```

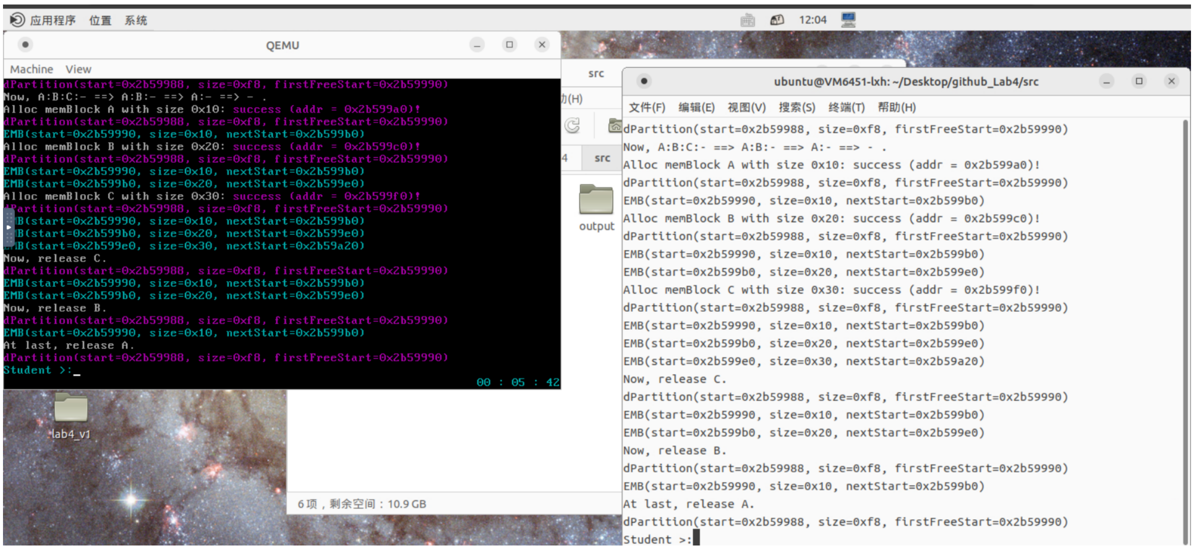
```
    } else myPrintf(0x7,"MALLOC FAILED, CAN't TEST dPartition\n");
}
```

testdP1先分配一块大小为0x100的空间，然后分配大小为0x10内存，成功，释放，分配0x20，成功，释放，分配0x40,成功，释放，分配0x80成功，释放，分配0x100，失败，原因是因为头结点和EMB块都需要占用一定的空间，所以不能把0x100空间全部分配出去



再来解释下testdP3:

testdP3是测试内存分配和空EMB的合并，按照`A:B:C:- ==> A:B:- ==> A:- ==> -NULL.`的顺序进行分配和释放，当分配完A,B,C三个内存节点的时候会出现四个EMB块，当释放C后由于前后链表都没有空的EMB块不能合并，故等待，当释放完B后可以和空的C链表合并，当释放A的时候又可以和上一次的空链表合并，合并完后发现仍然可以和dPHandler的EMB块合并，最终又恢复到malloc最初的状态，只有一个dPHandler和EMB块。（实验里面打印结果的时候我对于分配过内存但又free的EMB块没有打印出来)



部分结果滚动走了。。。

Bug解决

很多莫名其妙的bug，最终靠换了一个框架解决问题的(捂脸)。