

实验案例五：内核子系统—内存管理(指导文档)

实验案例五：内核子系统—内存管理(指导文档)

- 一、实验简介
- 二、实验内容步骤
 - 任务一: 打印空闲节点信息
 - 任务二: 动态内存分配策略修改

一、实验简介

内存管理模块管理系统的内存资源，它是操作系统的核心模块之一，主要包括内存的初始化、分配以及释放。在系统运行过程中，内存管理模块通过对内存的申请/释放来管理用户和OS对内存的使用，使内存的利用率和使用效率达到最优，同时最大限度地解决系统的内存碎片问题。鸿蒙内核LiteOS的内存池管理分为静态内存管理和动态内存管理，提供内存初始化、分配、释放等功能。其中静态内存管理主要由软件定时器模块使用，而动态内存用于管理系统的内核堆空间。

本实验基于 LiteOS-A 内核，旨在学习操作系统中静态内存与动态内存的概念及其在 LiteOS 内核中的应用。实验内容包括理解内核为何使用内存池、如何使用相关接口，并尝试修改动态内存管理代码，通过实践加深对 LiteOS 内核的理解。

二、实验内容步骤

本次实验需要依次完成下列实验要求

任务一: 打印空闲节点信息

LiteOS 提供了 `LOS_MemFreeNodeShow` 函数，用于打印动态内存池中空闲内存块的相关信息。然而，该函数输出较为简略，无法显示每个内存块的实际大小及起始地址。本实验要求在 `LOS_MemFreeNodeShow` 基础上，实现新函数 `LOS_MemFreeNodeDetailShow`，能够打印空闲内存块的具体大小和起始地址。实验承接第二节系统调用实验，最终实现效果应与第二节实验预期结果一致。

流程

本节任务需要在 `//kernel/liteos_a/kernel/base/mem/tlsf/los_memory.c` 中实现函数 `LOS_MemFreeNodeDetailShow`，打印内核堆中所有空闲块的详细信息。由于内核中实际上已经含有打印动态内存池中空闲块的函数 `LOS_MemFreeNodeShow`，因此本节任务 `LOS_MemFreeNodeDetailShow` 的实现可以直接参考 `LOS_MemFreeNodeShow`。

想要实现打印动态内存池中所有内存块，就必须遍历内存池中的空闲块链表，而在LiteOS的动态内存管理中，这一点是使TLSF算法实现的，所有的空闲内存块都挂在内存池结构体 `OsMemPoolHead` 的 `freeList` 上，而 `freeList` 数组的大小为 `OS_MEM_FREE_LIST_COUNT`。因此需要遍历 `OS_MEM_FREE_LIST_COUNT` 依次访问内存池链表数组 `freeList`。如果链表不为空，就说明存在空闲块。而储存内存块的大小信息的是内存块结构体 `OsMemNodeHead` 的成员 `sizeAndFlag`。使用宏定义 `OS_MEM_NODE_GET_SIZE()` 从该成员数据上取得内存块的大小并打印。具体代码如下：

```
UINT32 LOS_MemFreeNodeDetailShow(VOID *pool)
{
    struct OsMemPoolHead *poolInfo = (struct OsMemPoolHead *)pool;
```

```

if ((poolInfo == NULL) || ((UINTPTR)pool != (UINTPTR)poolInfo->info.pool)) {
    PRINT_ERR("wrong mem pool addr: %#x, line:%d\n", poolInfo, __LINE__);
    return LOS_NOK;
}

struct OsMemFreeNodeHead *node = NULL;
UINT32 countNum[OS_MEM_FREE_LIST_COUNT] = {0};
UINT32 index;
UINT32 intSave;

MEM_LOCK(poolInfo, intSave);
PRINTK("\n ***** left free node number*****\n");
for (index = 0; index < OS_MEM_FREE_LIST_COUNT; index++) { // 遍历内存池的空闲链
    node = poolInfo->freeList[index];
    if (node) PRINTK("\nfree index: %03u: \n", index);
    while (node) { // 依次遍历链表，不为空说明有空闲
        PRINTK("address: %p, size: %d\n", node, OS_MEM_NODE_GET_SIZE(node->header.sizeAndFlag)); //打印内存块地址和大小。
        node = node->next;
        countNum[index]++;
    }
}
PRINTK("\n *****\n\n");
MEM_UNLOCK(poolInfo, intSave);

return LOS_OK;
}

```

内存块

任务二: 动态内存分配策略修改

LiteOS-a在负责内核堆内存管理的动态内存池中，对于位于区间 $[2^7, 2^{31}]$ 的内存申请使用Good-Fit策略分配空闲内存块。因而如果在OpenHarmony的内核堆初始化后即函数 `oskHeapInit()` 中运行如下代码：

```

PRINTK("\n\n");
void *p1, *p2, *p3, *p4;
p1 = LOS_MemAlloc(m_aucSysMem0, (1 << 10) + (1 << 5));
p2 = LOS_MemAlloc(m_aucSysMem0, 24);
p3 = LOS_MemAlloc(m_aucSysMem0, 1 << 10);
p4 = LOS_MemAlloc(m_aucSysMem0, 1 << 10);

LOS_MemFree(m_aucSysMem0, p1);
LOS_MemFree(m_aucSysMem0, p3);
LOS_MemFreeNodeDetailShow(m_aucSysMem0);
p1 = LOS_MemAlloc(m_aucSysMem0, (1 << 10) + (1 << 5));
LOS_MemFreeNodeDetailShow(m_aucSysMem0);

```

分别申请内存块 $p1 = 2^{10} + 2^5$, $p2 = 24$, $p3 = 2^{10}$, $p4 = 2^{10}$ 的内存块，接着分别释放内存块 $p1$ 与 $p3$ ，再重新申请大小为 $2^{10} + 2^5$ 的内存块，并在过程中打印内存池中的空闲块信息,那么启动OHOS后应当会出现如下信息：

```

***** left free node number*****
free index: 055:
address: 0x4029f02c, size: 1036
address: 0x4029ebdc, size: 1068

free index: 138:
address: 0x4029f844, size: 1443760
*****

***** left free node number*****
free index: 055:
address: 0x4029f02c, size: 1036
address: 0x4029ebdc, size: 1068
free index: 138:
address: 0x4029fc70, size: 1442692
*****

```

由于采用 Good-Fit 策略，新申请的内存不会从原先释放的相同大小内存块中分配，而是将较大内存块拆分后分配更大一级的内存。本节实验要求阅读 `LOS_MemAlloc` 的实现方法，并尝试将原有的 Good-Fit 策略修改为 Best-Fit 策略。若修改正确，重新运行上述代码应打印如下信息：

```

***** left free node number*****
free index: 055:
address: 0x4029f02c, size: 1036
address: 0x4029ebdc, size: 1068

free index: 138:
address: 0x4029f844, size: 1443760
*****

***** left free node number*****
free index: 055:
address: 0x4029f02c, size: 1036

free index: 138:
address: 0x4029f844, size: 1443760
*****

```

流程

LiteOS动态内存申请通过 `LOS_MemAlloc` 完成，而在其中调用函数 `OSMemFindNextSuitableBlock` 来从内存池中的空闲内存块中找到能够满足当前申请大小的内存块，并将其从链表上摘下返回给调用者。因而对于内存空间分配策略是Good-Fit还是Best-Fit的关键，就在于函数 `OSMemFindNextSuitableBlock` 中是如何查找满足要求的内存块的。

而在原本的 `OSMemFindNextSuitableBlock` 中，Good-Fit实现的关键点就在于下图红框所示位置,其中`curIndex`是当前申请的内存大小恰好符合的链表索引。但是由于Good-Fit策略，这里是直接选择了 `curIndex+1` 赋值给 `index`，相当于直接选择了更大级别的内存块。

```

do {
    if (size < OS_MEM_SMALL_BUCKET_MAX_SIZE) {
        index = fl;
    } else {
        sl = OsMemSlGet(size, fl);
        curIndex = ((fl - OS_MEM_LARGE_START_BUCKET) << OS_MEM_SLI) + sl + OS_MEM_SMALL_BUCKET_COUNT;
        index = curIndex + 1;
    }

    tmp = OsMemNotEmptyIndexGet(poolHead, index);
    if (tmp != OS_MEM_FREE_LIST_COUNT) {
        index = tmp;
        goto DONE;
    }

    for (index = LOS_Align(index + 1, 32); index < OS_MEM_FREE_LIST_COUNT; index += 32) { /* 32: align size */
        mask = poolHead->freeListBitmap[BITMAP_INDEX(index)];
        if (mask != 0) {
            index = OsMemFFS(mask) + index;
            goto DONE;
        }
    }
} while (0);

```

除了上述问题之外，下图红框位置可以看到该函数直接将位于 `index` 位置的链表头的空闲内存块返回给调用者，而跳过了寻找合适内存块的过程，这在Good-Fit策略上是可以的。但是在Best-Fit策略则要求遍历链表，从中找出最合适能够满足需求的空闲块，因此下图的红框部分也同样需要修改。

```

1  DONE:
2      *outIndex = index;
3      return poolHead->freeList[index];
4

```

由于LiteOS中已经定义了 `OsMemFindCurSuitableBlock` 用于寻找最合适的内存块，因此可以直接使用，最后的 `OsMemFindNextSuitableBlock` 的实现代码如下：

```

STATIC INLINE struct OsMemFreeNodeHead *OsMemFindNextSuitableBlock(VOID *pool, UINT32 size,
UINT32 *outIndex)
{
    struct OsMemPoolHead *poolHead = (struct OsMemPoolHead *)pool;
    UINT32 fl = OsMemFlGet(size);
    UINT32 sl;
    UINT32 index, tmp;
    UINT32 curIndex = OS_MEM_FREE_LIST_COUNT;
    UINT32 mask;

    do {
        if (size < OS_MEM_SMALL_BUCKET_MAX_SIZE) {
            index = fl;
        } else {
            sl = OsMemSlGet(size, fl);
            curIndex = ((fl - OS_MEM_LARGE_START_BUCKET) << OS_MEM_SLI) + sl +
OS_MEM_SMALL_BUCKET_COUNT;
            // 任务二：修改为Best-Fit
            // *****开始*****
            index = curIndex;          // 选择当前索引
            // *****结束*****
        }

        tmp = OsMemNotEmptyIndexGet(poolHead, index);
        if (tmp != OS_MEM_FREE_LIST_COUNT) {

```

```

        index = tmp;
        goto DONE;
    }

    for (index = LOS_Align(index + 1, 32); index < OS_MEM_FREE_LIST_COUNT; index += 32)
    { /* 32: align size */
        mask = poolHead->freeListBitmap[BITMAP_INDEX(index)];
        if (mask != 0) {
            index = osMemFFS(mask) + index;
            goto DONE;
        }
    }
} while (0);

if (curIndex == OS_MEM_FREE_LIST_COUNT) {
    return NULL;
}

*outIndex = curIndex;
return osMemFindCurSuitableBlock(poolHead, curIndex, size);
DONE:
*outIndex = index;
// 任务二：修改为Best-Fit
// *****开始*****
return osMemFindCurSuitableBlock(poolHead, index, size); //调用
osMemFindCurSuitableBlock, 选择最合适的空闲块。
// *****结束*****
}

```

完成以上工作之后，即可在 `oskHeapInit` 函数下增加任务要求的代码，验证修改效果

```

m_aucSysMem0 = m_aucSysMem1 = ptr;
ret = LOS_MemInit(m_aucSysMem0, size);
if (ret != LOS_OK) {
    PRINT_ERR("vmm kheap_init LOS_MemInit failed!\n");
    g_vmBootMemBase -= size;
    return ret;
} else {
    // 实验任务：打印内存池信息
    // *****开始*****
    PRINTK("\n\n");
    void *p1, *p2, *p3, *p4;
    p1 = LOS_MemAlloc(m_aucSysMem0, (1 << 10) + (1 << 5));
    p2 = LOS_MemAlloc(m_aucSysMem0, 24);
    p3 = LOS_MemAlloc(m_aucSysMem0, 1 << 10);
    p4 = LOS_MemAlloc(m_aucSysMem0, 1 << 10);

    LOS_MemFree(m_aucSysMem0, p1);
    LOS_MemFree(m_aucSysMem0, p3);
    LOS_MemFreeNodeDetailShow(m_aucSysMem0);
    p1 = LOS_MemAlloc(m_aucSysMem0, (1 << 10) + (1 << 5));
    LOS_MemFreeNodeDetailShow(m_aucSysMem0);

    LOS_MemFree(m_aucSysMem0, p1);
    LOS_MemFree(m_aucSysMem0, p2);
    LOS_MemFree(m_aucSysMem0, p4);
    LOS_MemFreeNodeDetailShow(m_aucSysMem0);
    OsMemInfoPrint(m_aucSysMem0);
    // *****结束*****
}

```

最后重新编译运行OHOS即可得到目标结果。