

实验案例五：内核子系统—内存管理

实验案例五：内核子系统—内存管理

- 一、实验简介
- 二、实验内容及要求
 - 任务一: 打印空闲节点信息
 - 任务二: 动态内存分配策略修改
- 三、实验原理
 - 1. 静态内存
 - (1). 内存结构
 - (2). 内存管理
 - (3). 静态内存使用
 - 2. 动态内存
 - (1). 内存结构
 - (2). TLSF算法
 - (3). 内存管理
 - (4). 动态内存使用
- 四、实验流程
 - 任务一：打印空闲节点信息
 - 1. 流程
 - 任务二：动态内存分配策略修改
 - 1. 流程
- 五、参考资料

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、实验内容及要求

本次实验需要依次完成下列实验要求，并提交在qemu中运行的最终结果截图。

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。

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:

```
PRINTF("\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
*****
```

可以看到由于使用Good-Fit策略, 新申请的内存没有从原先释放的相等大小的内存中分配, 而是将大内存块内存块拆分, 直接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
*****

```

三、实验原理

1. 静态内存

(1). 内存结构

静态内存实质上是一个静态数组，静态内存池内的块大小在初始化时设定，初始化后块大小不可变更，每次内存块的申请和释放都以块大小为粒度。这使得其相比于动态内存能够有着更高的分配和释放效率，并且内存池中不会产生碎片。但是相对而言的，这也使得静态内存的分配无法按需申请，容易浪费空间。如图1所示，静态内存池由一个控制块和若干相同大小的内存块构成，控制块位于内存池头部，用于内存块管理。

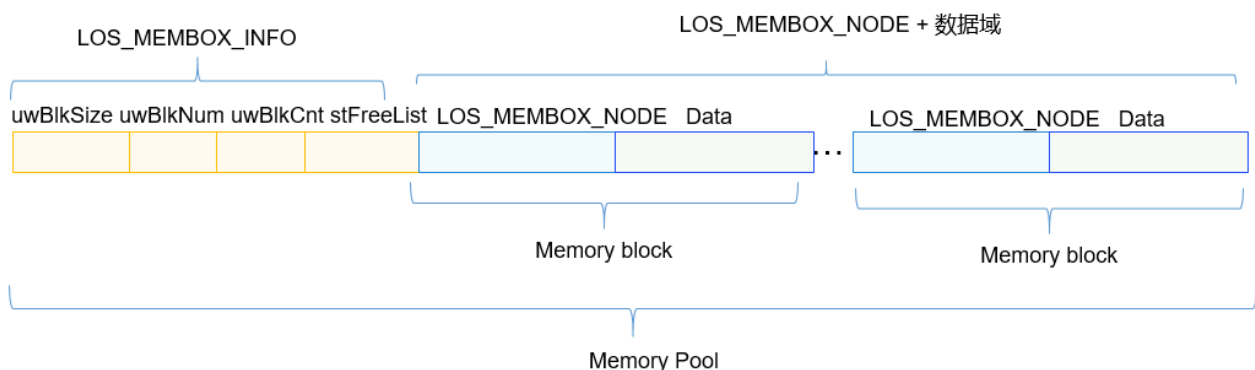


图1.静态内存管理结构[1]

在OpenHarmony的LiteOS-a内核之中，静态内存管理的声明代码位于 `//kernel/liteos_a/kernel/include/los_membox.h` 之中，涉及静态内存管理的数据结构信息如下：

```

typedef struct {
    UINT32 uwBlkSize;           /**< 块大小*/
    UINT32 uwBlkNum;           /**< 块数量 */
    UINT32 uwBlkCnt;           /**< 已分配的块数量 */
    LOS_MEMBOX_NODE stFreeList; /**< 空闲链表 */
} LOS_MEMBOX_INFO;

typedef struct tagMEMBOX_NODE {
    struct tagMEMBOX_NODE *pstNext; /**< 指向内存池中下一个空闲节点的指针 */
} LOS_MEMBOX_NODE;

```

`LOS_MEMBOX_INFO` 数据结构保存着LiteOS的静态内存池信息，位于内存池的头部，其中结构体成员作用如下：

- uwBlkSize: 每个静态内存块的所占空间大小。
- uwBlkNum: 静态内存池之中所包含的内存块数量。
- uwBlkCnt: 静态内存池中已经被分配的内存块数量。
- stFreeList: 空闲链表，静态内存池中所有空闲内存块连接在该链表上。

`LOS_MEMBOX_NODE` 起到连接空闲内存块的作用，其结构体成员信息如下：

- pstNext: 指向下一个空闲内存块的 `LOS_MEMBOX_NODE` 节点，同时也是内存块的起始地址。

(2). 内存管理

在静态内存管理之中，LiteOS实现了多个内存池管理函数，包括如下：

```
UINT32 LOS_MemboxInit(VOID *pool, UINT32 poolSize, UINT32 blkSize); // 内存池初始化
VOID *LOS_MemboxAlloc(VOID *pool); // 申请内存
UINT32 LOS_MemboxFree(VOID *pool, VOID *box); // 释放内存
VOID LOS_MemboxClr(VOID *pool, VOID *box); // 清空指定内存块
VOID LOS_ShowBox(VOID *pool); // 打印内存池详细信息
UINT32 LOS_MemboxStatisticsGet(const VOID *boxMem, UINT32 *maxBlk, UINT32 *blkCnt, UINT32 *blkSize); // 获取指定静态内存池的信息
```

上述函数的具体定义位于 `//kernel/liteos_a/kernel/base/mem/membox/los_membox.h` 文件中。本文仅对其中最关键的操作实现流程进行讲解，其余函数可自行查阅源码以深入了解。

初始化:

`UINT32 LOS_MemboxInit(VOID *pool, UINT32 poolSize, UINT32 blkSize)` 函数实现了静态内存池的初始化，其中参数 `pool` 是待初始化内存的起始地址，`poolSize` 是待初始化内存的大小，`blkSize` 则代表希望初始化得到的内存池中每个内存块大小。整个初始化流程大致如下：

1. 在 `pool` 所指向的地址处创建一个 `LOS_MEMBOX_INFO` 结构体 `boxInfo`。
2. 根据参数 `blkSize` 计算静态内存块大小，并赋值给 `boxInfo->uwBlkSize`。
3. 通过 `(poolSize - sizeof(LOS_MEMBOX_INFO)) / boxInfo->uwBlkSize` 表达式计算剩余待初始化内存空间可分割的内存块数量，记录在 `boxInfo->uwBlkNum` 中。
4. 初始化 `boxInfo` 中剩余的成员 `boxInfo->uwBlkCnt` 值为0，此时没有内存块被使用。
5. 最后初始化各个静态内存块节点 `LOS_MEMBOX_NODE`，将其连接起来，并最后挂在 `boxInfo->stFreeList` 链表上。

```
LITE_OS_SEC_TEXT_INIT UINT32 LOS_MemboxInit(VOID *pool, UINT32 poolSize, UINT32 blkSize)
{
    LOS_MEMBOX_INFO *boxInfo = (LOS_MEMBOX_INFO *)pool;
    LOS_MEMBOX_NODE *node = NULL;
    UINT32 index;
    ...
    boxInfo->uwBlkSize = LOS_MEMBOX_ALIGNED(blkSize + OS_MEMBOX_NODE_HEAD_SIZE);
    boxInfo->uwBlkNum = (poolSize - sizeof(LOS_MEMBOX_INFO)) / boxInfo->uwBlkSize;
    boxInfo->uwBlkCnt = 0;
    if (boxInfo->uwBlkNum == 0) {
        MEMBOX_UNLOCK(intSave);
        return LOS_NOK;
    }
}
```

```

}
node = (LOS_MEMBOX_NODE *) (boxInfo + 1);
boxInfo->stFreeList.pstNext = node;
for (index = 0; index < boxInfo->uwBlkNum - 1; ++index) {
    node->pstNext = OS_MEMBOX_NEXT(node, boxInfo->uwBlkSize);
    node = node->pstNext;
}
node->pstNext = NULL;
MEMBOX_UNLOCK(intSave);
return LOS_OK;
}

```

申请内存:

`VOID *LOS_MemboxAlloc(VOID *pool)` 函数实现静态内存块的申请，其中参数 `pool` 指明需要申请内存块的目标内存池。整个内存申请工作并不复杂。简单地从空闲链表 `stFreeList` 中寻找一个空闲内存块节点并将其链表上取下，更新内存池头信息 `boxInfo`，最后返回取下的内存块地址即可。

```

LITE_OS_SEC_TEXT VOID *LOS_MemboxAlloc(VOID *pool)
{
    LOS_MEMBOX_INFO *boxInfo = (LOS_MEMBOX_INFO *)pool;
    LOS_MEMBOX_NODE *node = NULL;
    LOS_MEMBOX_NODE *nodeTmp = NULL;
    UINT32 intSave;
    if (pool == NULL) {
        return NULL;
    }
    MEMBOX_LOCK(intSave);
    node = &(boxInfo->stFreeList);
    if (node->pstNext != NULL) {
        nodeTmp = node->pstNext;
        node->pstNext = nodeTmp->pstNext;
        OS_MEMBOX_SET_MAGIC(nodeTmp); // 设置魔法数字
        boxInfo->uwBlkCnt++;
    }
    MEMBOX_UNLOCK(intSave);
    return (nodeTmp == NULL) ? NULL : OS_MEMBOX_USER_ADDR(nodeTmp);
}

```

释放内存:

`UINT32 LOS_MemboxFree(VOID *pool, VOID *box)` 用于释放内存申请的静态内存块，其中参数 `pool` 指明所要释放的内存块所属的内存池，`box` 则为所要释放的内存块地址。函数首先重新为释放的内存块创建 `LOS_MEMBOX_NODE` 节点，接着将该内存块挂到内存池头 `boxInfo` 的空闲链表节点头上，最后更新内存池头其它信息。

```

LITE_OS_SEC_TEXT UINT32 LOS_MemboxFree(VOID *pool, VOID *box)
{
    LOS_MEMBOX_INFO *boxInfo = (LOS_MEMBOX_INFO *)pool;
    UINT32 ret = LOS_NOK;
    UINT32 intSave;
    if ((pool == NULL) || (box == NULL)) {

```

```

        return LOS_NOK;
    }
    MEMBOX_LOCK(intSave);
    do {
        LOS_MEMBOX_NODE *node = OS_MEMBOX_NODE_ADDR(box);
        if (OsCheckBoxMem(boxInfo, node) != LOS_OK) {
            break;
        }
        node->pstNext = boxInfo->stFreeList.pstNext;
        boxInfo->stFreeList.pstNext = node;
        boxInfo->uwBlkCnt--;
        ret = LOS_OK;
    } while (0);
    MEMBOX_UNLOCK(intSave);
    return ret;
}

```

(3). 静态内存使用

在 LiteOS 中，静态内存池的使用频率较低。通过在内核源码中搜索 `LOS_MemboxAlloc` 可发现，只有 `u3g.c` 与 `los_swtmr.c` 两处使用该函数，其中 `u3g.c` 用于 USB 3G 数据卡支持，与本节内容无关，因此不做讨论。在 LiteOS-A 内核中，静态内存池主要用于软件定时器，其初始化操作仅在 `SwtmrBaseInit` 函数中执行。初始化所需的内存区域是通过系统内核堆动态申请获得的。

2. 动态内存

(1). 内存结构

动态内存管理主要用于用户需要使用大小不等的内存块的场景。当用户需要使用内存时，可以通过操作系统的动态内存申请函数索取指定大小的内存块，一旦使用完毕，通过动态内存释放函数归还所占用内存，使之可以重复使用。

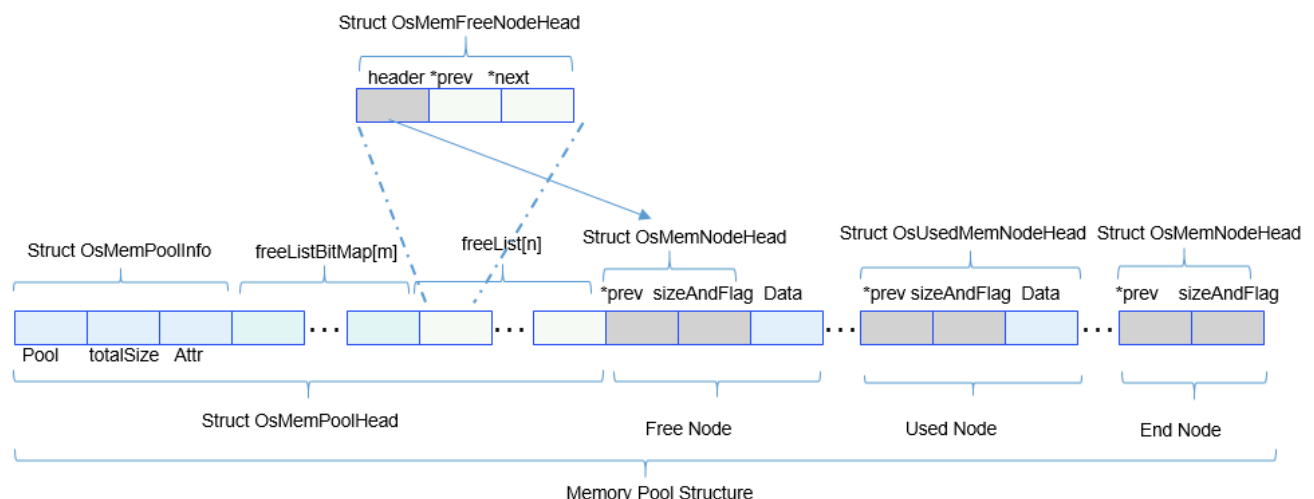


图2.动态内存管理结构[1]

如图2所示，动态内存池管理数据结构同样分为内存池池头与内存池节点两种，其声明位于 `//kernel/liteos_a/kernel/base/mem/tlsf/los_memory.c` 之中，详细信息如下：

内存池池头：

内存池池头部分包含内存池信息、位图标记数组和空闲链表数组。内存池信息包含内存池起始地址及堆区域总大小，内存池属性。位图标记数组有7个32位无符号整数组成，每个比特位标记对应的空闲链表是否挂载空闲内存块节点。空闲内存链表包含223个空闲内存头节点信息，每个空闲内存头节点信息维护内存节点头和空闲链表中的前驱、后继空闲内存节点。

```
struct OSMemPoolHead {
    struct OSMemPoolInfo info;
    UINT32 freeListBitmap[OS_MEM_BITMAP_WORDS];    // OS_MEM_BITMAP_WORDS = (((31 + (24 << 3)) >> 5) + 1) = 7
    struct OSMemFreeNodeHead *freeList[OS_MEM_FREE_LIST_COUNT]; //OS_MEM_FREE_LIST_COUNT=(31 + (24 << 3))=223
    ...
};
```

`OSMemPoolHead` 负责保存动态内存池信息以及内存块使用情况，一般位于初始化内存池的头部，其中的重要的数据成员作用如下：

- `info`: 负责记录动态内存池的基本信息，包括内存池起始地址，大小以及其它属性。
- `freeListBitmap`: TLSF算法使用的二级位图，记录各个区间内存的可用情况，共有 $7 * 32 - 1 = 223$ 个区间。
- `OSMemFreeNodeHead`: 空闲节点链表数组，大小为223，分别连接着位于223个区间的空闲内存块。

```
struct OSMemPoolInfo {
    VOID *pool;
    UINT32 totalSize;
    UINT32 attr;
    ...
};
```

`OSMemPoolInfo` 负责保存动态内存池基本信息以及内存块使用情况，一般位于初始化内存池的头部，其中的重要的数据成员作用如下：

- `pool`: 为动态内存池的起始地址。
- `totalSize`: 动态内存池大小。
- `attr`: 内存池属性。

内存池节点：

内存池节点共有3种类型，分别是未使用空闲内存节点，已使用内存节点和尾节点。每个内存节点维护一个前序指针，指向内存池中上一个内存节点，还维护内存节点的大小和使用标记。空闲内存节点和已使用内存节点后面的内存区域是数据域，尾节点没有数据域。

```
struct OSMemNodeHead {
    UINT32 magic;
    union {
        struct OSMemNodeHead *prev; /* The prev is used for current node points to the previous node */
    }
};
```

```

        struct OSMemNodeHead *next; /* The next is used for last node points to the expand
node */
    } ptr;
    ...
    UINT32 sizeAndFlag;
};

struct OSMemUsedNodeHead {
    struct OSMemNodeHead header;
    ...
};

struct OSMemFreeNodeHead {
    struct OSMemNodeHead header;
    struct OSMemFreeNodeHead *prev;
    struct OSMemFreeNodeHead *next;
};

```

其中的重要的数据成员作用如下:

- magic: 用于防止内存的越界使用。其位于结构体头部，因而一旦有操作越界修改了此块的内存，那么magic字段就会被修改，因而能够通过检测magic的一致性来判断内存的越界使用与否。
- header: 内存块节点。记录每个内存块的基本信息，包括内存块的大小、标志位以及指针。
- ptr: 内存块节点内部的联合体指针，一般用于内存块合并和分割。一般情况下只使用prev指针指向前一个内存块节点开始地址，而由于内存一般是连续的，因而下一个内存块的起始地址可通过 当前内存块地址 + sizeAndFlag 得到。而next指针只有在内存池中的最后一个节点，也就是哨兵节点会使用，一般情况下指向NULL。如果开启内存池扩展功能，新分配的内存块可能并不连续，因而next指针此时会指向扩展内存块起始地址。
- prev, next: 分别指向前一个与后一个内存块起始地址。但是在 OSMemNodeHead 和 OSMemFreeNodeHead 中不一样的是，OSMemNodeHead 中的prev指向虚拟地址上连续的上一块内存。而 OSMemFreeNodeHead 节点中的prev和next指针用于指向前一个与后一个空闲的内存块可能并不连续，两个连接的空闲内存块之间可能包含已经被分配使用的内存。
- sizeAndFlag: 内存块的大小以及标志位。由于认为LiteOS的动态内存最大可分配内存不大于 2^{27} ，因此只需要28位即可表示内存块的大小。剩余的4位则用作标志位包含如下几种：

```

#define OS_MEM_NODE_USED_FLAG      0x80000000U ///< 已使用标签
#define OS_MEM_NODE_ALIGNED_FLAG  0x40000000U ///< 对齐标签
#define OS_MEM_NODE_LAST_FLAG     0x20000000U /* Sentinel Node | 哨兵节点标签*/
#define OS_MEM_NODE_ALIGNED_AND_USED_FLAG (OS_MEM_NODE_USED_FLAG | OS_MEM_NODE_ALIGNED_FLAG | OS_MEM_NODE_LAST_FLAG)

```

在获取内存块大小时通过宏 OSMEM_NODE_GET_SIZE(sizeAndFlag) 取得内存块大小。

(2). TLSF算法

相比静态内存，动态内存管理的优势在于可以按需分配，但其缺点是容易产生内存碎片。为此，OpenHarmony LiteOS-A 在 **TLSF** 算法基础上对内存区间划分进行了优化，以提升分配性能并降低碎片率。

TLSF(Two-Level Segregate Fit)算法是一种用于实时操作系统的内存分配算法，其将空闲块按照大小分级，形成不同块大小范围的分级(class)，组内空闲块用链表链接起来。每次分配先按分级大小范围查找到相应链表，再从相应链表挨个检索合适的空闲块，如果找不到，就在大小范围更大的一级查找，直到找到合适的块分配出去。在此之上，采用两级的位图来管理不同大小范围的空闲块链。

- **第一级位图：**存储粗粒度范围的内存块链的**空闲**状态，一般是2的幂次粒度。如图中的红色框所示。
- **第二级位图，**存储一级位图中的每一位对应内存块组的一项，表示内存块的细粒度范围。如图中的蓝色框所示。

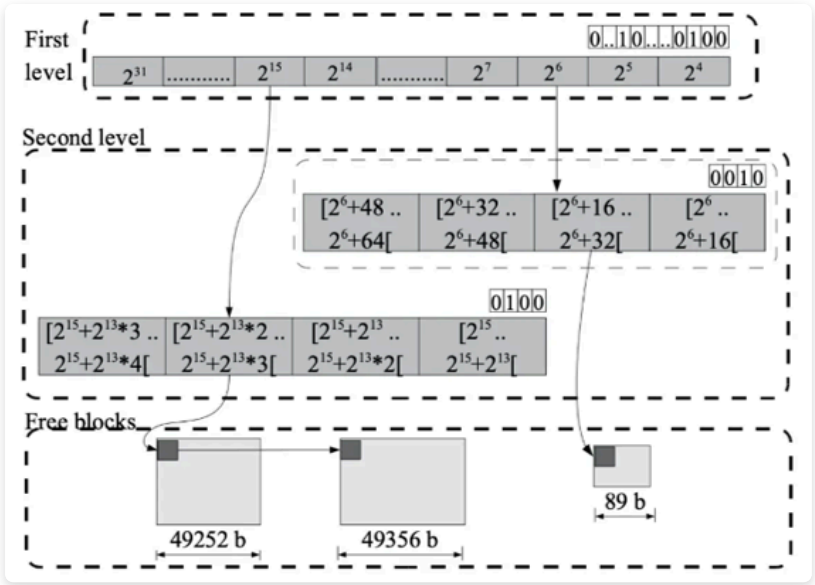
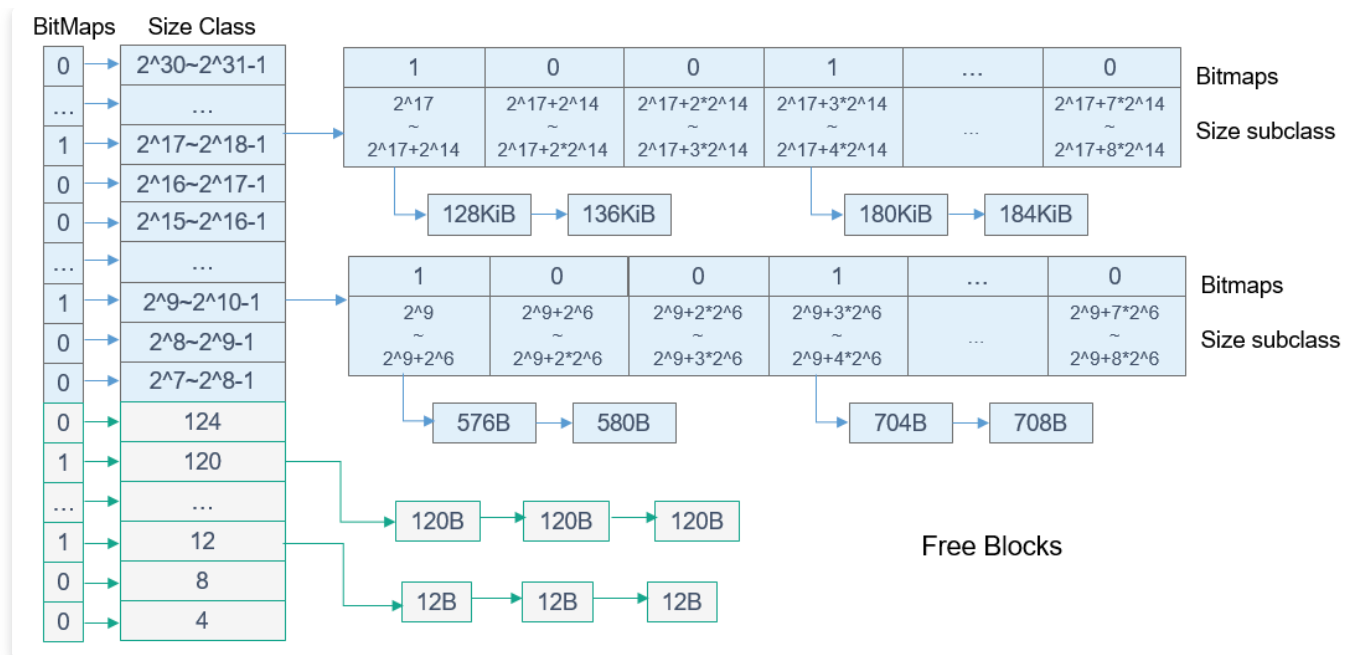


图3.二级位图示意图

如OpenHarmony官方文档的堆内存小节中所述,在LiteOS-a实现的TLSF大致实现如下:

根据空闲内存块的大小，使用多个空闲链表来管理。根据内存空闲块大小将其分为两个部分： $[4, 127]$ 和 $[2^7, 2^{31}]$ ，如图4中的**Size Class**所示：

1. 对 $[4, 127]$ 区间的内存进行等分，如图4下半部分所示，分为31个小区间，每个小区间对应内存块大小为4字节的倍数。每个小区间对应一个空闲内存链表和用于标记对应空闲内存链表是否为空的一个比特位，值为1时，空闲链表非空。 $[4, 127]$ 区间的31个小区间内存对应31个比特位进行标记链表是否为空。
2. 大于127字节的空闲内存块，按照2的次幂区间大小进行空闲链表管理。总共分为24个小区间，每个小区间又等分为8个二级小区间，见图4上半部分的Size Class和Size SubClass部分。每个二级小区间对应一个空闲链表和用于标记对应空闲内存链表是否为空的一个比特位。总共 $24 \times 8 = 192$ 个二级小区间，对应192个空闲链表和192个比特位进行标记链表是否为空。



在 LiteOS-a 中上述 $[4, 127]$ 内存块一共分为了 31 个小区间，而 $[2^7, 2^{31}]$ 则分成了 24 个一级小区间以及 8 个二级小区间，因而整个内存池共有 $31 + 24 * 8 = 223$ 个区间，这一划分对应于 `osMemPoolHead` 内存池头结构体中的 `freeListBitmap` (`OS_MEM_BITMAP_WORDS` 共 7 个字节，224 位) 和 `freeList` (`OS_MEM_FREE_LIST_COUNT` 共 223 个链表)。这些设计正是 LiteOS 中 **TLSF** 算法实现的基础。

(3). 内存管理

LiteOS-a中负责申请动态内存块的函数为 `LOS_MemAlloc`,其定义位于 `//kernel/liteos_a/kernel/base/mem/tlsf/los_memory.c` 之中。正如上文所说, LiteOS的动态内存池分为两个区域, 分别是[4, 127]按照31等分的内存区间以及由 $[2^7, 2^{31}]$ 分成的24个一级小区间以及8个二级小区间。对于这两个区间, LiteOS采用了不同的内存分配策略,

- $[4, 127]$ 区间：由于 LiteOS 中分配的动态内存块必须按照 4 字节对齐，因而在此区间分配的内存必定为 4 的整数倍，可以看到图 4 中位于该区间的链表所指向的内存块都是同样的大小。故而用户在动态内存池中申请大小位于 $[4, 127]$ 区间的内存时会被对齐到 4 的倍数，LiteOS 的内存分配也天然的就是使用 Best-Fit 策略，每次用户申请的内存都一定能够得到完美匹配的申请大小的内存块，不会产生内存碎片。
- $[2^7, 2^{31}]$ 区间：LiteOS 对于用户申请的大小位于此区间的内存时采用 Good-fit 策略，每次寻找满足用户申请大小的内存块时都是直接从更大一级寻找，以此减少内存检索时间。具体实现位于函数

`OsMemFindNextSuitableBlock` 中, 代码如下:

```

STATIC_INLINE struct OsMemFreeNodeHead *OsMemFindNextSuitableBlock(VOID *pool, UINT32 size,
UINT32 *outIndex)
{
    ...
    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;    // 当前索引位+1, Good-Fit
        }
    } while (1);
}

```

```

    }
    ...
}while(0);
...
*outIndex = index;
return poolHead->freeList[index];
}

```

(4). 动态内存使用

在 LiteOS-A 中，本节介绍的动态内存技术主要用于内核堆的管理。在系统运行过程中，堆内存管理模块通过对内存的申请与释放管理用户态和内核态对内存的使用，以优化内存利用率并最大限度地降低碎片产生。在源码中，LiteOS-A 堆内存的初始化路径为：`OsMain` → `OsSysMemInit` → `OsKHeapInit` → `LOS_MemInit`，可依此顺序查阅实现代码。动态内存初始化时，堆大小为 512 KB，但由于需要按照 MB 对齐，实际初始化的堆空间约为 0x1617cc（约 1.38 MB）。内核堆空间虽不大，但系统运行过程中所有内存申请均来源于此，包括上一节提及的静态内存池所使用的内存区域。此外，通过 `malloc` 函数申请的小于 16 KB 的内存均来自内核堆；而大于 16 KB 的内存则通过伙伴算法分配连续的物理页帧，该机制将在下一节中详细介绍。

四、实验流程

任务一：打印空闲节点信息

本节任务需要在 `//kernel/liteos_a/kernel/base/mem/tlsf/los_memory.c` 中实现函数

`LOS_MemFreeNodeDetailShow`，其能打印内核堆中所有空闲块的详细信息，并在系统初始化内核堆的函数 `OsKHeapInit` 中添加代码，使其完成堆初始化后打印空闲块信息。

1. 流程

1. 自行阅读 `los_memory.c` 中代码，理解 LiteOS-a 内核如何实现动态内存管理。
2. 参考 `LOS_MemFreeNodeShow` 的实现，在 `los_memory.c` 中添加函数 `LOS_MemFreeNodeDetailShow`，并在内核堆初始化函数 `OsKHeapInit` 中使用新添加的函数打印空闲块信息。
3. 编译并进入 OHOS 系统，查看运行效果。

任务二：动态内存分配策略修改

本节任务需要学习 LiteOS 动态内存池针对用户通过 `LOS_MemAlloc` 的内存申请需求而分配内存空间的策略由 Good-Fit 修改为 Best-Fit。

1. 流程

1. 学习 `LOS_MemAlloc` 的实现方法，了解 LiteOS 是如何使用 Good-Fit 分配策略。
2. 修改 `OsMemFindNextSuitableBlock` 函数，使其实现 Best-Fit 分配策略。
3. 将实验文档任务二中所给代码添加至 `oskHeapInit` 中，并重新编译运行系统验证结果。

五、参考资料

[1]. 内存管理模块:<https://docs.openharmony.cn/pages/v4.1/zh-cn/device-dev/kernel/kernel-small-basic-memory-heap.md>