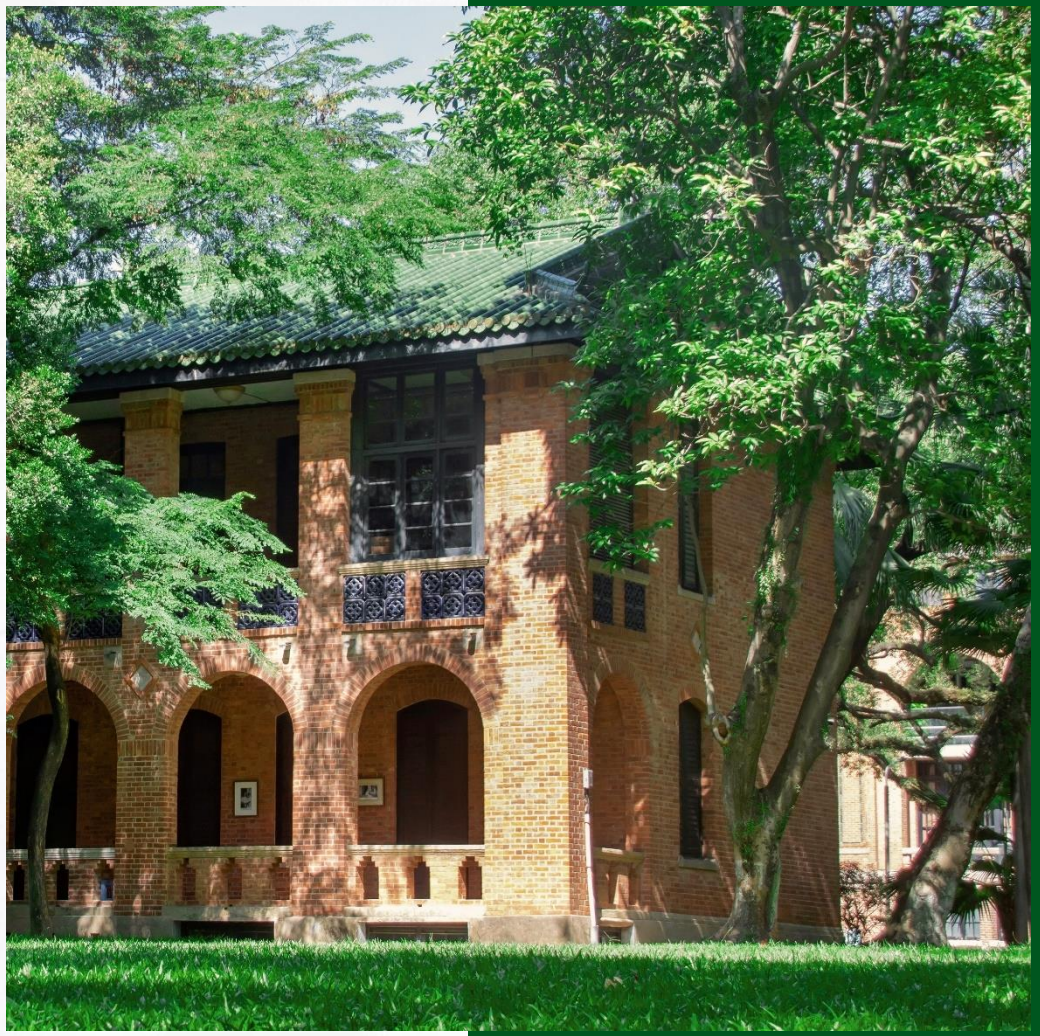
中山大學
SUN YAT-SEN UNIVERSITY

鸿蒙操作系统
内存管理



HarmonyOS的内存管理方案

内存管理模块：管理系统的内存资源，主要包括对内存的**初始化**、**分配**和**释放**，可以分为动态内存管理和静态内存管理。

1. **静态内存管理：**在静态内存池中分配用户初始化时预设固定大小的内存块。
   - 优点：分配和释放效率高，静态内存池中无碎片。
   - 缺点：只能申请到初始化预设大小的内存块，不能按需申请。
2. **动态内存管理：**在动态内存池中分配用户指定大小的内存块。
   - 优点：按需分配。
   - 缺点：内存池中可能会出现碎片。

LiteOS 静态内存管理（MemoryBox机制）

1. 静态内存实质上是一个静态数组，其大小在初始化时设定，初始化后块大小不可变更。
2. 静态内存池由一个控制块和若干相同大小的内存块组成。
   - 控制块位于内存池头部，用于内存块管理，包含内存块大小BlkSize，内存块数量BlkNum，已分配使用的内存块数量BlkCnt和空闲内存块链表FreeList。
   - 内存块的申请和释放以**块大小**为粒度，每个内存块包含指向下一个内存块的指针pstNext。
3. 内存的总容量 = 内存块大小 + 内存池控制块大小。

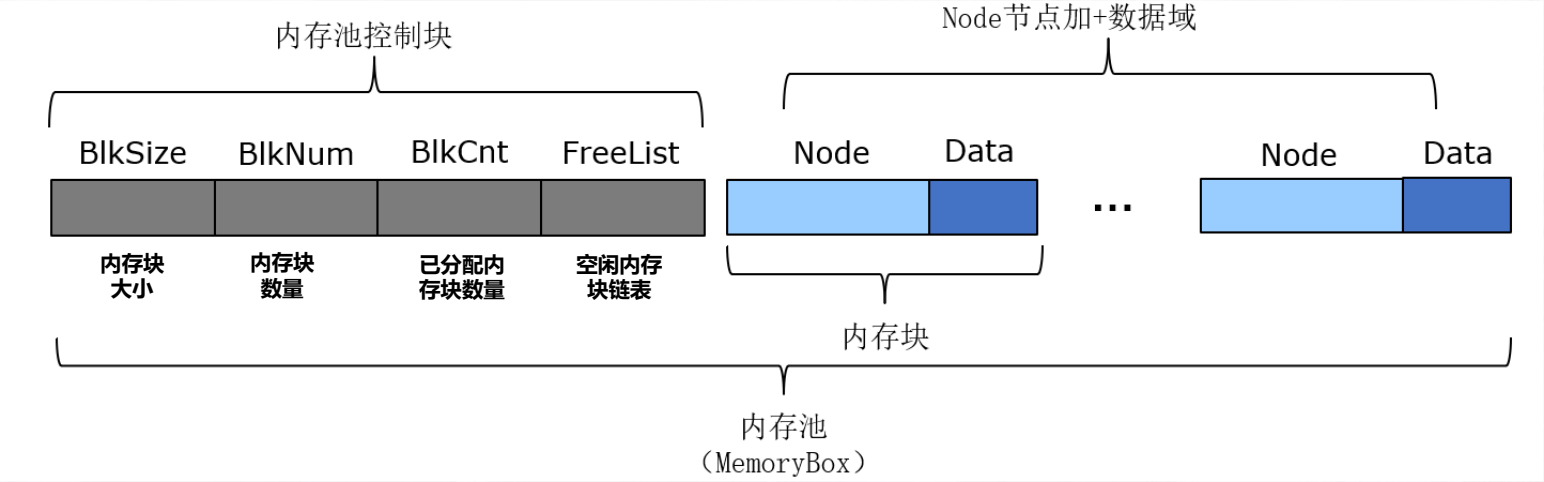


图6-1LiteOS静态内存结构

- **LiteOS 静态内存管理（MemoryBox机制）**

  LiteOS 静态内存开发流程

  1. 规划静态内存池
  2. 内存池初始化　(LOS_MemboxInit)
  3. 申请内存块　　(LOS_MemboxAlloc)
  4. 处理内存块　　(LOS_MemboxStatisticsGet，LOS_ShowBox)
  5. 清除内存内容　(LOS_MemboxClr)
  6. 释放内存　　(LOS_MemboxFree)

LiteOS-m 动态内存管理

1. LiteOS-m使用bestfit（最佳适配）内存管理算法。
2. Bestfit内存管理结构：
   - LosMemPoolInfo：记录内存池的起始地址和总大小。
   - LosmultipleDlinkHead：数组的每一个元素都是双向链表，用于记录不同大小的**空闲块组**。例如，第n个双向链表存放大小为$[2^n, 2^{n+1}]$的内存块。
   - LosmemDyNode：存放各个节点（内存块）。

<table>
<tr><th colspan="2">LosMemPoolInfo</th><th colspan="3">LosmultipleDlinkHead</th><th colspan="4">LosmemDynNode</th></tr>
<tr><td rowspan="2">Start Address</td><td rowspan="2">Size</td><td>Pre</td><td>...</td><td>Pre</td><td rowspan="2">First Node</td><td rowspan="2">Second Node</td><td rowspan="2">...</td><td rowspan="2">End Node</td></tr>
<tr><td>Next</td><td>...</td><td>Next</td></tr>
</table>

图6-2 LiteOS-m动态内存结构

LiteOS-m 动态内存管理

3. 每次申请内存时，先从DlinkHead数组中检索大小**最合适的空闲内存块进行分配**，同时将该内存块对应的节点从双向链表中删除。
4. 每次释放内存时，会将该内存块作为空闲节点**重新插入到对应大小**的双向链表中，以便下次再次使用。
5. 实际上的LiteOS的动态内存管理更为复杂，上面只是bestfit的基本数据结构。为了获得更优的性能，降低了碎片率， LiteOS的动态内存管理在TLSF算法的基础上，对区间的划分进行了优化。

<table>
<tr><td colspan="2">LosMemPoolInfo</td><td colspan="3">LosmultipleDlinkHead</td><td colspan="4">LosmemDynNode</td></tr>
<tr><td rowspan="2">Start Address</td><td rowspan="2">Size</td><td>Pre</td><td>...</td><td>Pre</td><td rowspan="2">First Node</td><td rowspan="2">Second Node</td><td rowspan="2">...</td><td rowspan="2">End Node</td></tr>
<tr><td>Next</td><td>...</td><td>Next</td></tr>
</table>

图6-2 LiteOS-m动态内存结构

LiteOS-m 动态内存管理

TLSF算法[1]（全称Two-Level Segregated Fit，内存两级分割策略算法）

1. 传统的动态内存分配算法的缺点
   - 时间复杂度高，算法寻找合适内存块的时间不确定。
   - 碎片化问题。
2. TLSF算法的三个设计目标
   - 降低 worst-case 内存分配时间。
   - 降低内存分配时间。
   - 减少内存碎片和内存浪费。
3. TLSF算法的三大组成部分
   - 分级空闲块链表（Segregated Free List）
   - 二级位图（Two-Level Bitmap）
   - Good-fit 策略

1. Masmano M, Ripoll I, Crespo A, et al. TLSF: A new dynamic memory allocator for real-time systems[C]//Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004. IEEE, 2004: 79-88.

LiteOS-m 动态内存管理

常见的空闲块链表结构有**隐式链表**、**显式链表**和**分级空闲块链表**。

1. 隐式链表（管理所有内存块）

- 链接所有内存块，只记录内存块大小，由于内存块紧密相连，通过头结点指针加**内存块大小**即可得到下一个内存块的位置。由于没有显式指明内存块的地址，而是通过计算得到，所以又叫做隐式链表。
- 当需要分配内存时，需要从第一块内存块开始检索，检查该内存块是否被分配以及内存块大小是否被满足，直到找到大小合适的空闲块分配出去。检索的时间复杂度最大。

2. 显式链表（只管理空闲块）

- 显式空闲块链表在空闲块头部添加一个指针域，**指向下一个空闲块**，这样检索时会**跳过已分配的内存块**。

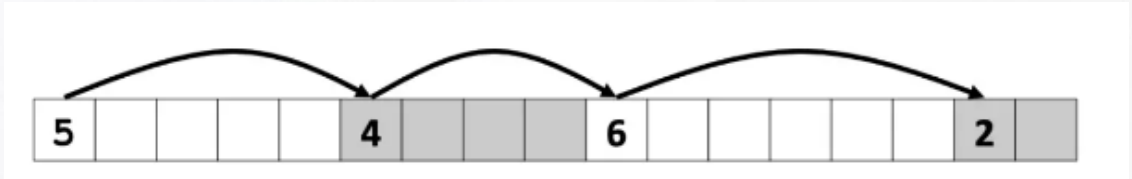


图6-3 隐式链表

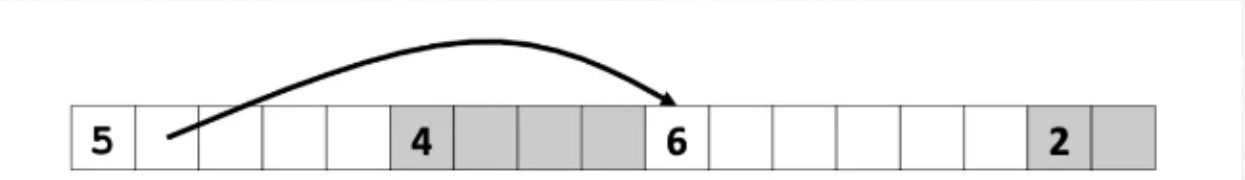


图6-4 显式链表

隐式链表和显式链表的检索速度均为O(n)，这在实时操作系统中是难以应用的

LiteOS-m 动态内存管理

分级空闲块链表（Segregated Free List）

- 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。
- 分级空闲块链表的检索速度能被优化到O(log n)。

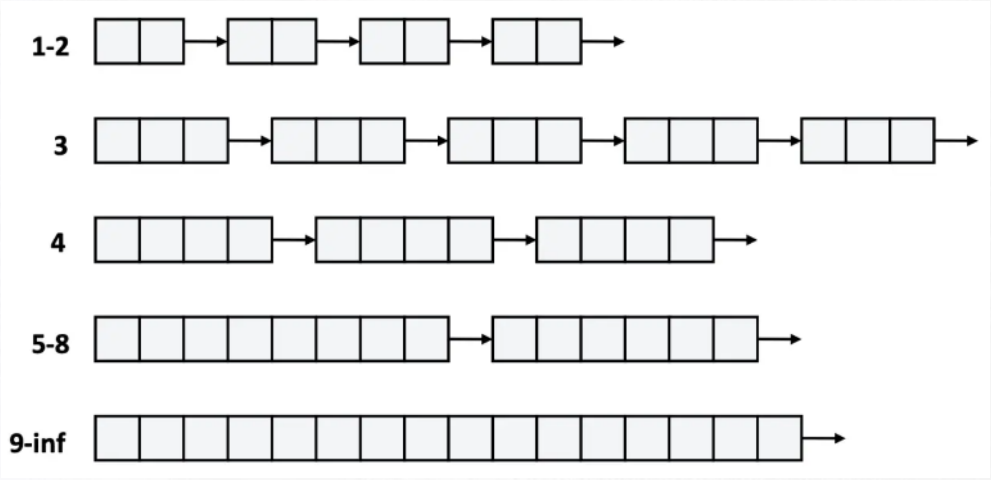


图6-5 分级空闲块链表

LiteOS-m 动态内存管理

二级位图（Two-Level Bitmap）

1. **位图**是用来跟踪**已分配**的存储块的一种结构。
   - 用1bit表示对应的存储块的分配状态，当值为1时表示该存储块已经被分配，反之表示该存储块是空闲的。
   - 位图具有节省存储空间，检索速度快的优势。
2. **TLSF**采用了两级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组的**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。

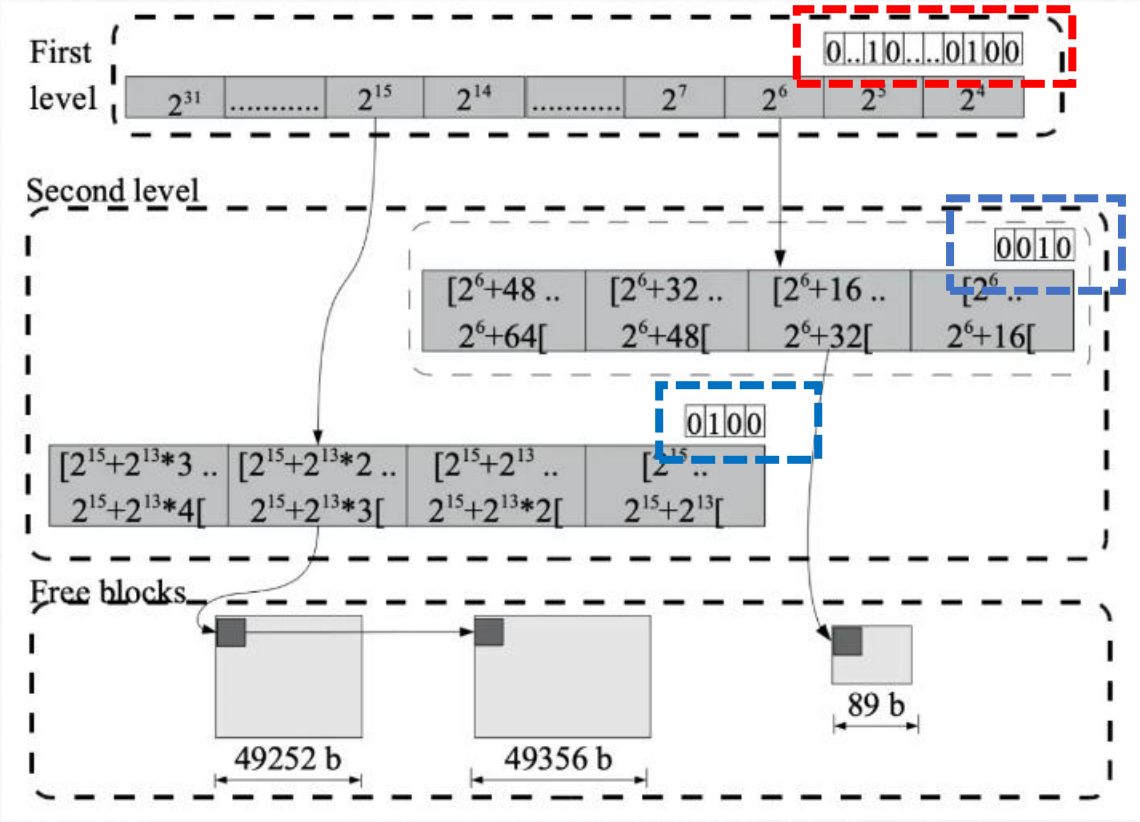


图6-6 二级位图示意图

LiteOS-m 动态内存管理

Good-fit 策略（以申请大小为**69**的内存块为例）

1. Best-fit（内部碎片最优）

- 通过位图找到请求大小所在的第一级位图对应的粗粒度范围[64~128]。
- 在粗粒度范围[64~128]内，根据二级位图索引检索第二级位图得到细粒度范围[68~70]。
- 最后遍历[68~70]对应的空闲块链表，找到最佳的空闲块69。

2. Good-fit 策略（速度最优）

- 与Best-fit的区别是在检索二级位图时，直接选取刚好比69大的下一个区间，即[71~73]，然后将其中任意一个空闲块直接分配。
- 这样不需要第三步的遍历过程，因此能达到一个O(1)的检索速度。

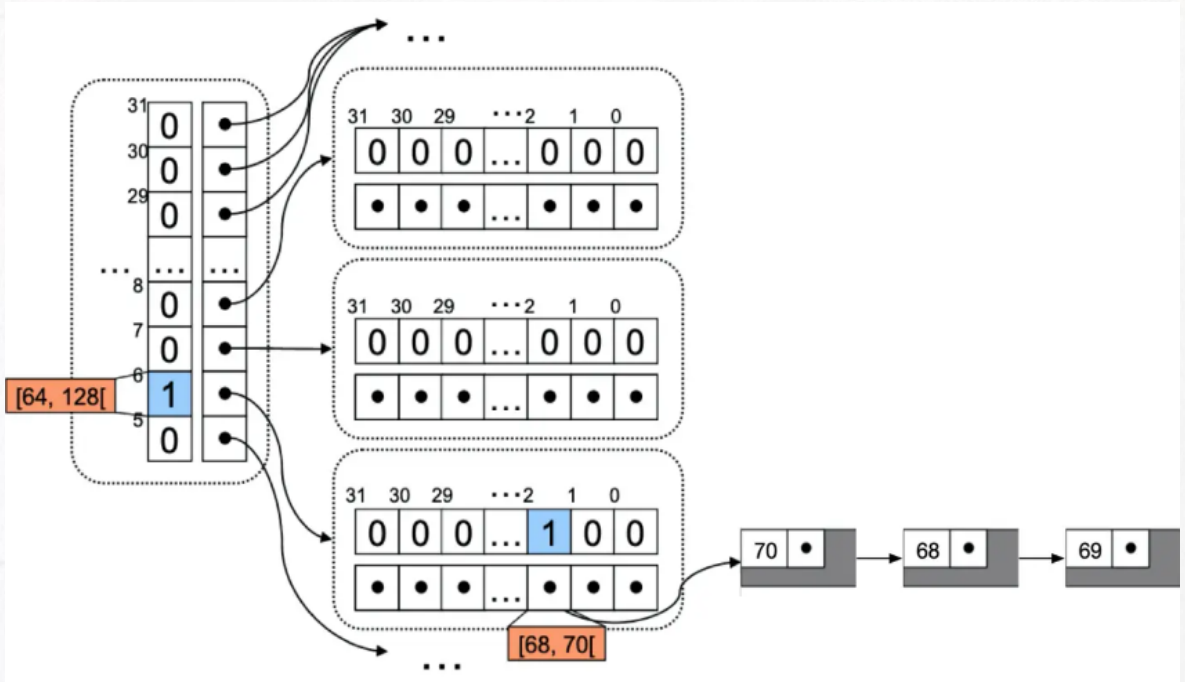


图6-7 Best-fit策略示意图

- **LiteOS-m 动态内存管理**

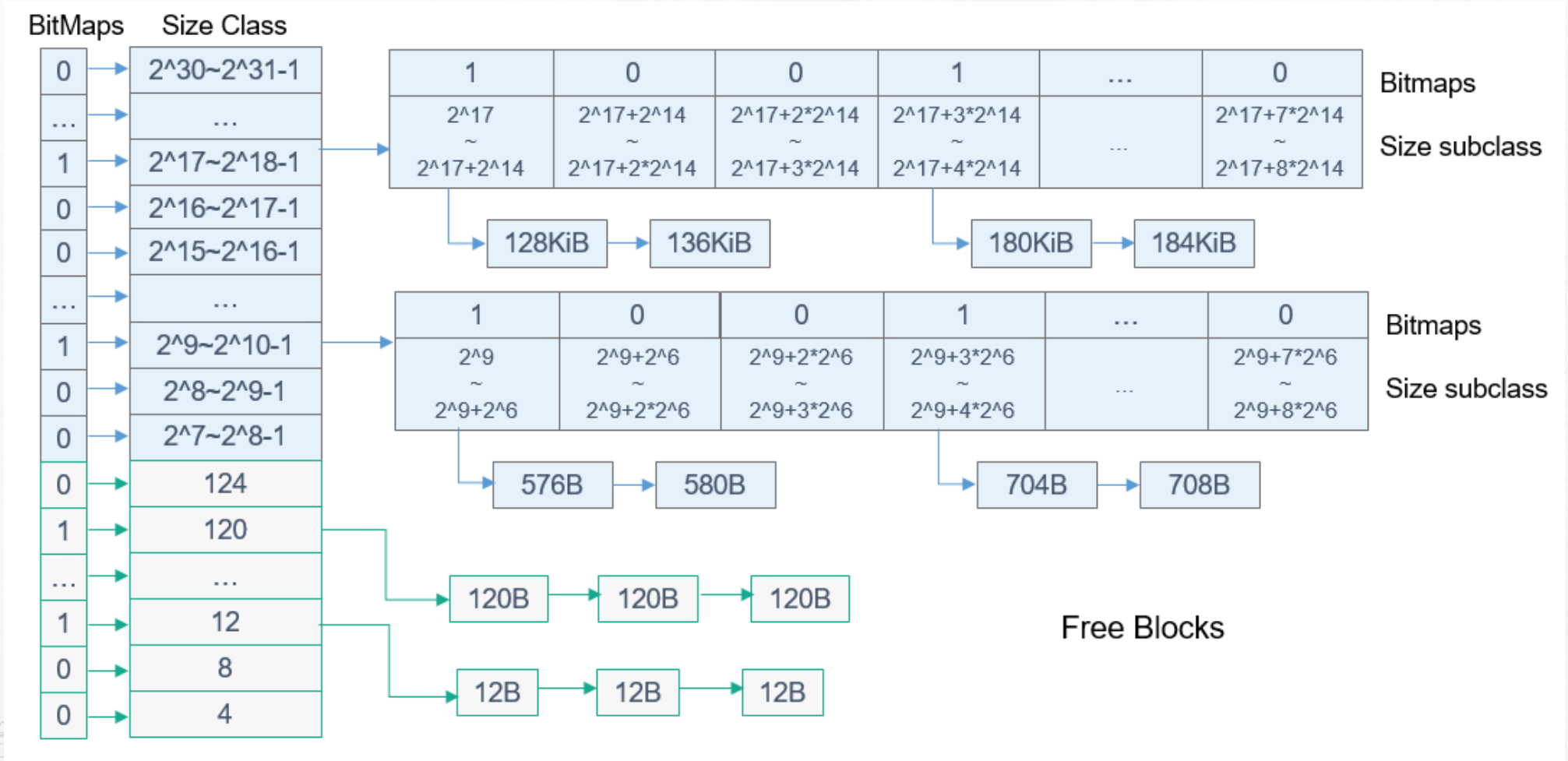


图6-8 LiteOS动态内存管理框架

LiteOS-m 动态内存管理

1. 根据空闲内存块的大小，LiteOS-m使用多级空闲链表来管理，同时将内存空闲块大小分为两个部分：[4, 127（ 2^7 -1）]和[2^7, 2^31]。其中，[4, 127]部分**直接使用空闲链表**存储对应大小的空闲块，[2^7, 2^31]部分使用**二级位图**来管理。

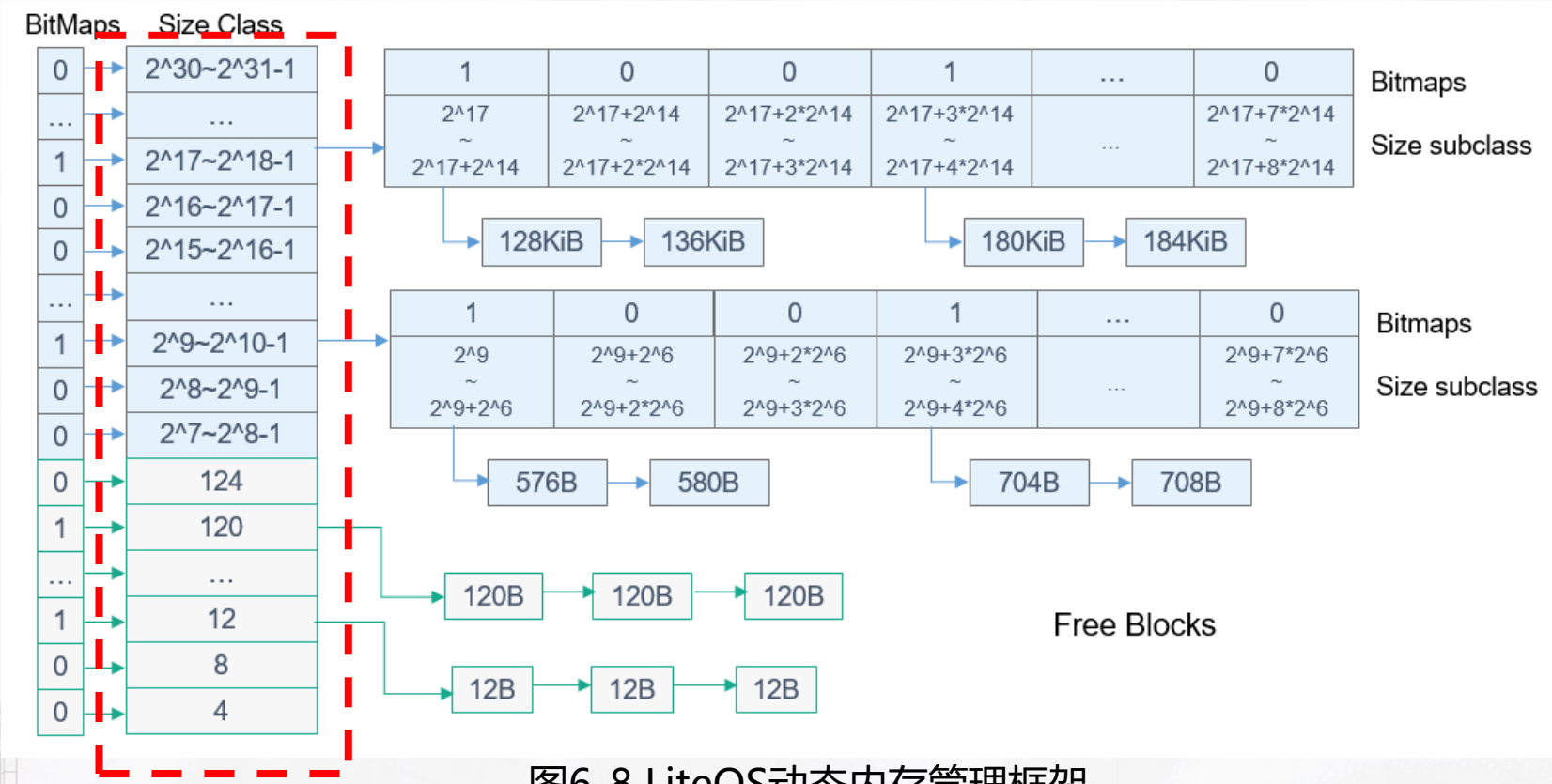


图6-8 LiteOS动态内存管理框架

LiteOS-m 动态内存管理

2. [4,127]区间的内存进行等分，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。

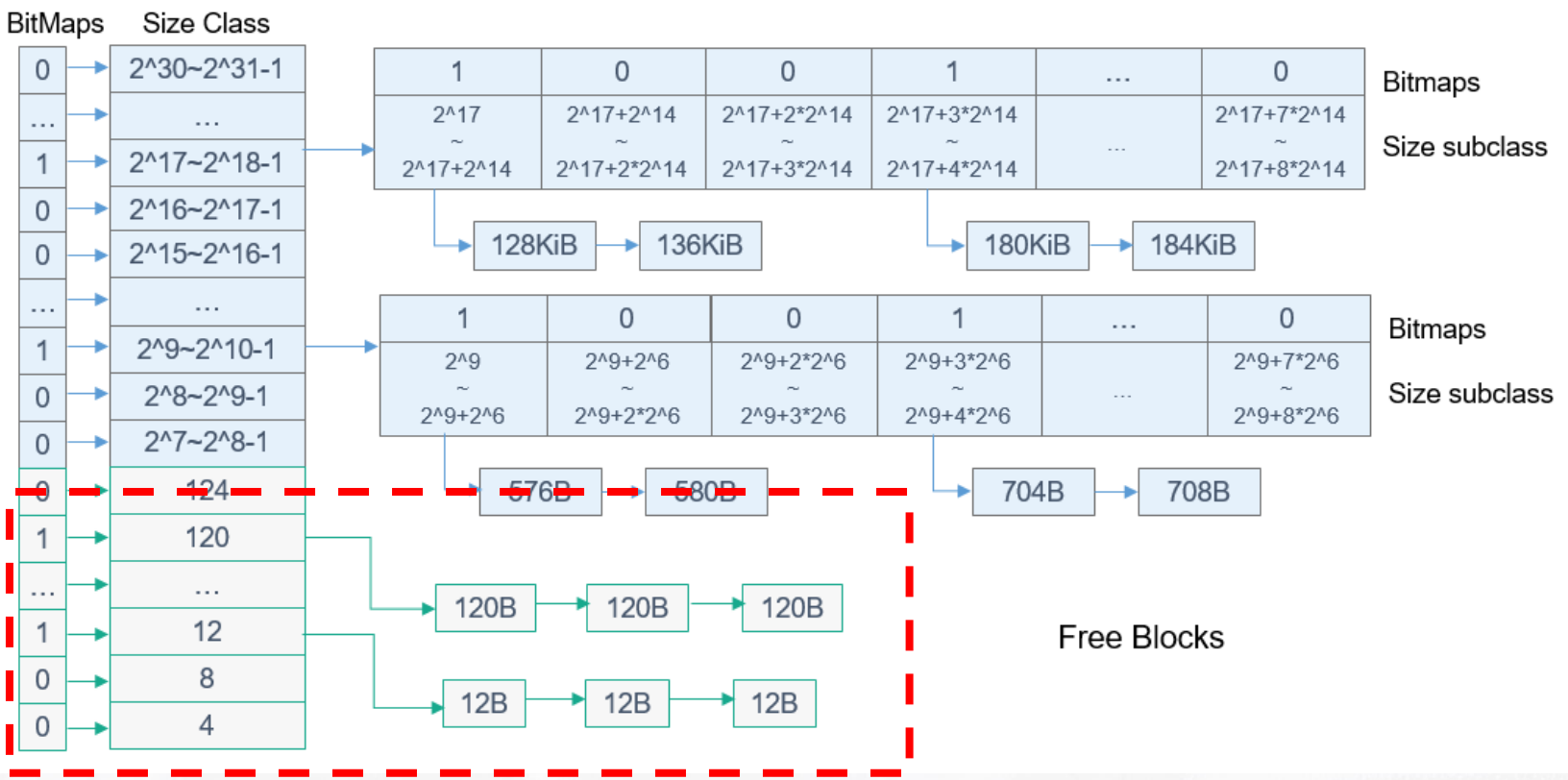


图6-8 LiteOS动态内存管理框架

- **LiteOS-m 动态内存管理**

3. 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**。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比特位**。总共24*8=192个二级小区间，对应192个空闲链表和192个比特位。

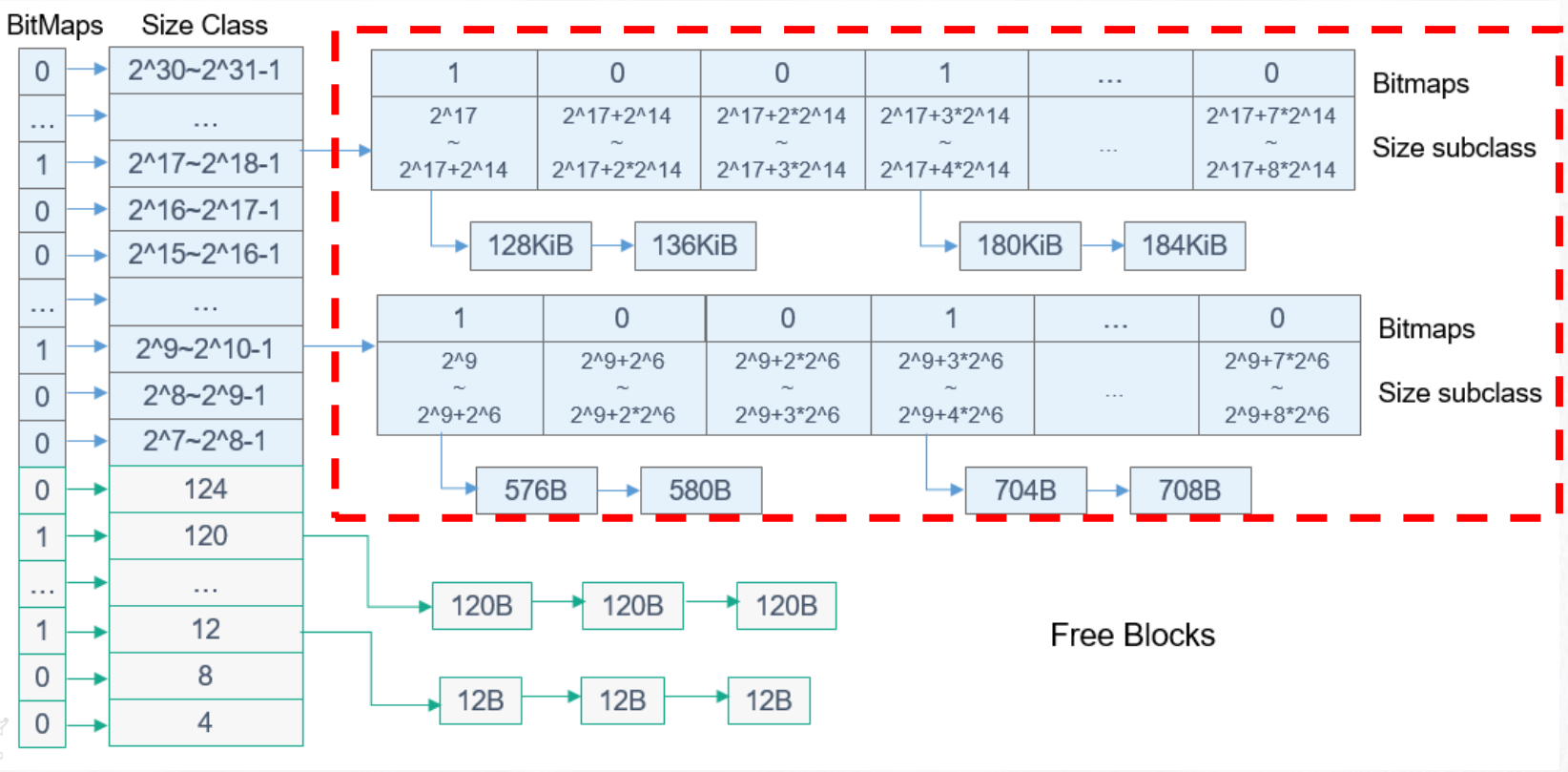


图6-8 LiteOS动态内存管理框架

LiteOS-a 动态内存管理

1. LiteOS-a除了支持bestfit算法外，还支持bestfit_little算法。
2. bestfit_little算法是在bestfit算法的基础上加入slab机制。
3. Bestfit算法在每次分配内存时，都会选择内存池中**最小最适合的内存块**进行分配，而slab机制用于限定分配固定大小的内存块，从而减小产生内存碎片的可能。
4. LiteOS-a中的slab机制支持配置slab class的数目以及每个class的最大空间。

<table>
<tr><th colspan="4">内存池头部</th><th>Slab class区域</th><th>内存池区域</th></tr>
<tr><td>节点头指针</td><td>节点尾指针</td><td>内存总大小</td><td>Slab Class控制结构</td><td>Slab class机制管理的内存，LosHeapNode链表进行控制，同时被OslabMem数据控制</td><td>除掉slab class以外剩余的内存池部分，按bestfit算法进行分配管理</td></tr>
</table>

图6-9 slab机制的内存结构

LiteOS-a 动态内存管理

slab机制示例：

1. 假设内存池中共有4个slab class，每个slab class的最大空间为512字节。
   - 第一个slab class被分为32个16字节的slab块。
   - 第二个slab class被分为16个32字节的slab块。
   - 第三个slab class被分为8个64字节的slab块。
   - 第四个slab class被分为4个128字节的slab块。
2. 这四个slab class是从内存池中按照最佳适配算法分配的，每次申请的内存块的大小为512字节。

LiteOS-a 动态内存管理

slab机制示例：

1. **初始化：**首先初始化内存池，然后在初始化后的内存池中按照最佳适配算法申请4个slab class，接着逐个按照slab内存管理机制初始化4个slab class。
2. **申请：**每次申请内存，先在满足申请大小的最佳slab class中申请（如用户申请20字节的内存，就在slab块大小为32字节的slab class中申请）。
3. **分配与回收：**申请成功，将slab内存块整块返回给用户时，释放内存时也整块进行回收。

特殊情况：

1. 如果满足条件的slab class中已无可以分配的内存块，则从内存池中按照最佳适配算法申请，而不会继续从有着更大slab块空间的slab class中申请。
2. 释放内存时，先检查释放的内存块是否属于slab class，如果是则将其还回对应的slab class中，否则放回内存池中。

中山大學
SUN YAT-SEN UNIVERSITY

谢谢观看

SUN YAT-SEN UNIVERSITY