# 内存管理模块

内存管理模块主要管理系统中内存资源，包括物理内存与进程逻辑地址空间两部分。物理内存的管理主要是对空闲物理内存的管理，物理内存和交换区的分配、释放算法等。

由于内存管理部件的差异，这一模块是Unix V6与Unix V6++改动较大的模块。Unix V6++中使用页目录和页表模拟了Unix V6中所使用的2组16个内存管理寄存器(KISA0 ~ KISA7、UISA0 ~ UISA7)，页大小由64个字节变成4K，每个进程允许的最大逻辑地址空间设定为8M，远大于Unix V6的进程空间大小。同时保持原有Unix V6中进程图像整体换入换出，并不使用缺页中断。

## 物理内存管理

### Unix V6++物理内存组织结构

操作系统负责管理系统中所有可用物理内存资源，这也意味着它可以根据设计需要决定如何划分内存，划分出哪些不同区域以及每个区域大小多少。

在Unix V6++系统初始化阶段对物理内存进行区域的划分，分出不同的内存区域范围分别提供给内核和用户程序。并且使用三个管理不同内存区域范围的管理器，其主要功能是负责各自区域范围中内存资源分配及释放，这将在后面具体解释。以一台具有16M内存的机器为例，下图描述了物理内存区域划分情况(图5.1)。



#### 图5.1 Unix V6++物理内存组织结构

* **物理内存的0-1M区域**

物理内存的0-1M区域为保留区域，该区域包含了I386体系结构规定的显存与ROM BIOS占用的内存地址范围(640KB – 1MB)。对于此区域，我们系统中仅有用到向显存区域写入输出信息，并不将其作为内核进行内存资源管理的一部分，换言之，是弃之不用的。

* **物理内存的1M-1.5M区域**

物理内存的1M-1.5M区域用于存放操作系统内核映像，包括操作系统代码与数据。系统初起时，Bootloader将内核从磁盘装载到物理内存1M起始地址，所以内核在整个操作系统生命周期中始终占据该内存区域。

内核映像占据大小在编译期间已经确定，并不会随着系统运行而变化，所以不可能超出1.5M区域的限制。事实上，Unix V6++的内核映像远没有到0.5M这么大，仅占据0.5M区域中上百KB内存空间而已。

* **物理内存的1.5M-2M区域**

物理内存的1.5M-2M区域用于内核堆，我们在内核中重载了new/delete操作符用于分配/释放内核对象，而创建新的内核对象所要内存资源便是在这一区域中分配。

* **物理内存的2M-4M区域**

物理内存的2M-4M区域为内核页区域，用于满足内核数据的物理内存页需求，该区域的内存以页为单位分配，主要用于分配进程所使用的页表。

* **物理内存的4M-16M区域**

从4M-16M(或内存尾端)区域为用户页区域，是为用户进程准备的物理内存页，所有进程图像相关的如代码段、数据段、用户栈段以及ppda区占据内存都从该区域中分配。

可以看到，整个物理内存大致上可分成用于操作系统内核的0-4M物理内存，以及用于用户进程的4M-16M(或内存尾端)物理内存两种不同用途的区域。然而，无论是内核对象区、内核页区域还是用户页区域，都是被内核“管”起来的，即由内核而不是用户程序负责内存的分配和释放。

操作系统0-4M内存被映射到逻辑地址空间中的0xc0000000-0xc0400000。因此，内核页区域和内核对象区域是可以直接被操作系统内核访问的，而用户页区域在被进行页表映射之前是无法被访问的。

### 内存管理类

系统中的物理内存资源按照用途不同被划分成内核对象区、内核页区域以及用户页区域这三种区域，而我们对这三个不同区域中内存资源的管理使用的都是同一算法­。

在Unix V6++中有三个内存管理类分别负责内核对象区、内核页区域以及用户页区域的物理内存管理，分别是KernelAllocator，KernelPageManager与UserPageManager。

其中，KernelAllocator管理内核对象区域的内存分配和释放，在内核中使用new操作符进行对象动态分配的内存由该类管理。KernelPageManager管理内核区域内的物理内存，这些物理页主要用于新创建进程的页表。UserPageManager管理用户页区域的物理内存，这些物理页存储进程图像数据。这三个内存管理类之间的关系如下图所示(图5.2)。



#### 图5.2 内存管理类结构

UserPageManager与KernelPageManager都是分配物理页，继承自基类PageManager，区别仅在于对基类中Initialize()函数的不同实现，从而定位到各自管理内存区域的起始位置和长度。

IAllocator是一个接口，将物理内存分配算法抽象为Alloc()与Free()两个接口函数，所有的内存分配类引用该算法接口分配内存，使用了Strategy模式。在Unix V6++中仅实现了一种FirstFitAllocator算法，三个不同的内存管理类都使用该算法。

### 内存管理的数据结构

以KernelPageManager来说，它包含成员变量MapNode map[]数组，这是一个记录空闲内存区的数组，数组的大小为常量MEMORY\_MAP\_ARRAY\_SIZE = 512。数组中的每一项都是一个MapNode类型的结构体，其定义如下：

struct MapNode

{

unsigned long m\_Size;

unsigned long m\_AddressIdx;

};

#### 代码5.1 MapNode定义

其中m\_Size是该空闲内存区的长度，单位是4K大小的物理页；而当MapNode数据结构用于盘交换区资源管理时，m\_Size则是空闲盘交换区的长度，以扇区为单位。m\_AddressIdx是相应空闲区的起始地址，单位同上。

各空闲内存区按照起始地址由低到高依次记录在空闲内存区数组MapNode map[]中，数组中每一项对应一组空闲内存页，随着内存资源分配和释放，内核不断修改MapNode map[]数组，使其始终保持着空闲内存资源的准确信息。

m\_Size为0的项为空白项，集中在MapNode数组的后面，这表示前面部分的数组元素已经能够描述区域中的全部空闲内存块，因而后面存在没有与之对应的空项，数组的最后一个项一定是空白项。

KernelPageManager::Initialize()函数用于初始化该内存管理类的对象，定位到其负责管理的区域，也就是2M-4M范围。

int KernelPageManager::Initialize()

{

PageManager::Initialize(); /\* 清空MapNode数组各项 \*/

this->map[0].m\_AddressIdx = 0x200000 / 0x1000;

this->map[0].m\_Size = (0x40000 – 0x200000) / 0x1000;

return 0;

}

#### 代码5.2 KernelPageManager::Initialize()函数

KernelPageManager::Initialize()函数首先调用基类的PageManager::Initialize()函数清空MapNode数组各项为零，然后使用数组第0项记录整个2M-4M的空闲内存区域，m\_AddressIdx中为空闲内存区起始地址，以4K页为单位；m\_Size存放空闲内存区大小，同样是 以页为单位。

### 内存资源分配算法

对空闲内存区的分配使用首次适应算法(First Fit)。该算法主要过程是：根据对空闲内存区大小的需求，从空闲内存区表的第一项开始逐项扫描，查找到第一个含有足够大空闲内存块的表项，如果本次内存需求恰好占用该项管理的整个空闲内存资源，则表中后续各项都向前移动一项；否则，根据该项中本次分配后剩余的数量，更新该项记录的空闲内存资源信息。

下面的图5.3给出了首次适应算法的一个示例。第一幅图中给出了新内存分配请求到达之前空闲内存区的状态，内存中有三块空闲内存块，占据了记录空闲内存区信息的MapNode map[]数组的前3项。

分配请求一：分配120单位大小内存。内核依次查找MapNode map[]数组中记录的空闲内存区信息，第0、1两项没有足够内存，直至查到第2项才有足够的空闲内存可供分配，然后更新该数组项中空闲内存区起始地址和大小信息，由[300, 212]变为[420, 92]，如图中所示(图5.3-a)。

分配请求二：分配100单位大小内存。同样，内核依次查找空闲内存区表各个记录项，结果这次在扫描第0项时即发现有足够内存，于是在该项中分配，而本次分配恰好占用该项管理的整个空闲内存资源，于是删除该记录项，并且需要把MapNode map[]数组中随后各项都向前移动一项，更新后空闲内存区数组信息如图中所示(图5.3-b)。

**512**

**512**

**0**

80

100

200

50

300

212

0

0

0

0

0

0

起始地址

大小

**80**

**200**

**180**

**250**

**300**

**分配前空闲内存状态**

**512**

80

100

200

50

**420**

**92**

0

0

0

0

0

0

起始地址

大小

**80**

**200**

**180**

**250**

**300**

**a) 分配120单位大小内存后**

新分配单元

**420**

**0**

**200**

**50**

**300**

**212**

0

0

0

0

0

0

0

0

起始地址

大小

**80**

**200**

**180**

**250**

**300**

**b) 分配100单位大小内存后**

新分配单元

**0**

#### 图5.3 首次适应算法(First Fit)示例

Unix V6++中，使用一个名为Allocator的类对内存分配算法进行封装，该类实现了Alloc()和Free()两个接口，其中Alloc()函数中运用的便是First Fit算法进行内存资源的分配。

class Allocator

{

public:

unsigned long Alloc(MapNode map[], unsigned long size);

unsigned long Free(MapNode map[], unsigned long size, unsigned long addrIdx);

};

#### 代码5.3 Allocator内存分配算法类

这里给出了Allocator类的定义(代码5.3)，Allocator::Alloc()函数用于分配内存，其接受2个参数，MapNode类型的数组以及本次请求分配内存的大小。

前面说过，Unix V6++中有三个内存管理类KernelAllocator，KernelPageManager和UserPageManager，它们各自负责不同的区域内核对象区(1.5M-2M)、内核页区域(2M-4M)以及用户页区域(4M-16M或内存尾端)，而对这三个不同区域中内存资源的管理使用的都是相同的算法。

之所以能做到这一点，关键在于这三个内存管理类拥有各自的空闲内存区表，也就是各自拥有独立的MapNode map[]数组记录着不同内存区域空闲内存情况；而Allocator::Alloc()函数接收MapNode类型的数组作为参数，使得各个内存管理类可以在调用Alloc()函数时将各自空闲内存区map[]数组传递给Allocator算法类，依照First Fit或其它算法执行分配操作。换言之，Allocator算法类的作用是将内存分配算法从三个内存管理类中抽象出来，使得我们更容易使用其它内存分配算法取而代之。

## 进程逻辑地址空间

### Unix V6++进程逻辑地址空间组织结构

在I386体系结构上，在理论上进程可以访问0-4G的逻辑地址空间，但Unix V6++对进程可访问逻辑地址空间做了如下的限制：每个进程的4G地址空间被分为用户态空间0-3G以及3G以上核心态空间。内核仅使用0xC0000000-0xC0400000区域，为4M大小逻辑地址空间，即只使用一张页表映射，对于用户区，现规定最多可使用8M的数据，即逻辑地址空间0x00000000-0x00800000区域，使用两张页表进行映射(图5.4)。



#### 图5.4 进程地址空间组织结构

每个进程的用户态空间(即0x00000000-0x00800000)是独立的，通过每个进程拥有的2张页表，映射到不同的物理内存上(除共享正文段有例外，可能会有多个进程共享)。而核心态地址空间是所有进程共享的，所有进程的核心态地址空间(即0xC0000000-0xC0400000)使用相同的页表映射，从而实现进程之间核心态地址空间的共享。

**. . .**

**. . .**

**0**

**1**

页目录

进程

页表0

进程

页表1

内核

页表

0x00000000

0x00400000

0x00800000

0x00400000

0xC0000000

0xC0400000

进程页表0

进程页表1

**上台进程**

**768**

新上台进程填写用户空间页表

#### 图5.5 系统中的页目录和页表

如图5.5所示，整个操作系统仅使用一张页目录表，页目录表总共有1024项，而Unix V6++仅使用其中的第0、1和768项，不使用其余项进行映射。该页目录表的第768项指向系统内核页表，而第0项与第1项指向进程的页表。

之所以确定使用页目录的第768项，是因为内核的逻辑地址在0xC0000000-0xC0400000范围，其32位地址的最高10比特的值为768，这个值是分页机制地址转换时页目录项的索引，所以要映射0xC0000000处的地址必须经过页目录的第768项。

在操作系统整个生命周期中，系统页目录和内核页表自从系统初始化完成后将始终保持不变。在进程切换时，需要对2张进程页表进行更新，新上台进程根据相对地址映照表填入进程页表，建立对上台进程的用户态空间映射。

### 用户态内存空间与MemoryDescriptor类

进程用户态下内存空间包括代码段，数据段，以及栈段。在Unix V6++的设计中，进程的ppda区（包括User结构与核心栈），用户态代码段与数据段在物理内存中是连续分配的，其中ppda区将在进程管理中详细介绍。代码段因为其只读特性，多个进程可以共享代码段，因此被分开存储。如图5.6所示，是两个进程物理内存的使用情况。



#### 图5.6 进程逻辑地址空间到物理地址空间的映射

各个进程都有自己独立的逻辑空间，可以保证访问不会冲突。然而整个系统的物理内存是由多个进程共享的，所以每个进程各自的页表作用就是将相同的逻辑地址映射到不同的物理地址上。

Unix V6++中使用MemoryDescriptor类来管理进程的用户态空间，以及建立用户态空间映射所需的相对地址映照表，如图5.7所示MemoryDescriptor的类图。



#### 图5.7 MemeoryDescriptor类

该类描述了正文段，数据段的起始地址与长度，同时还描述了栈与堆区的大小。成员变量m\_UserPageTableArray指向了该进程用户空间的2张页表，总共2048个页表项。

### MemoryDescriptor类与User结构

Unix V6++中使用MemoryDescriptor类封装原先user结构中的内存管理相关字段(如u\_uisa[]、u\_uisd[]、u\_tsize、u\_dsize、u\_ssize等)，因此在Unix V6++中把一个MemoryDescriptor类型的对象作为User类中的成员变量，用于替代原有的变量。

由于原来的Unix V6只需要管理64K用户态地址空间，因此只需要使用u\_uisa[]与u\_uisd[]两个数组的相对地址映照表就可以保存进程内存映射情况，但Unix V6++中每个进程使用2张页表即8K大小来管理内存映射，就不能像Unix V6中将内存映射信息放入user结构中，毕竟整个ppda区(包括User结构与核心栈)总共才占据一页内存，4K大小，不可能像包含u\_uisa[]、u\_uisd[]两个数组那样包含2张页表。因此需要为每个进程的2张页表额外分配内存空间，同时由MemoryDescriptor类中的成员变量m\_UserPageTableArray指向它们，每个进程根据自己user中的MemoryDescriptor找到对应页表。

使用MemoryDescriptor类管理所有内存映射情况，包括进程各数据段信息皆被从user结构中移入该类中。若今后对内存管理有相应的改动，例如扩大内存映射范围，从2张页表到4张页表，则可以使改动所牵涉到的类最少。

### MemoryDescriptor类的实现

class **MemoryDescriptor**

{

public:

/\* 创建进程时用于申请2张用户空间页表\*/

void Initialize();

/\* 在进程终止时，调用该函数释放进程拥有的页表 \*/

void Release();

/\* 建立用户态空间的相对地址映照表\*/

bool EstablishUserPageTable(int t\_addr, int t\_size, int d\_addr, int d\_size, int s\_size );

/\* 根据相对地址映照表以及正文段、ppda首址设置系统中2张用户态空间页表 \*/

void MapToPageTable();

};

#### 代码5.4 MemoryDescriptor类

MemoryDescriptor类中，Initialize()函数在创建进程时为2张用户空间页表分配所需内存空间，并建立MemoryDescriptor对象与新分配页表之间的指向关系。Release()函数则在进程终止时释放进程拥有的页表。

EstablishUserPageTable()函数用于建立用户态空间的相对地址映照表。这里需要重点解释的是：Initialize()函数在创建进程时分配的2张用户空间页表，虽然名为页表，实际上是作为相对地址映照表用途使用的，这样做是为了模仿原来Unix V6的user结构中u\_uisa与u\_uisd两个数组保存的相对地址映照表，2张页表中存放的都是相对地址，即假设正文段、ppda区首地址为0而构成的。

MapToPageTable()函数则是完成从相对地址映照表到实际页表的装填工作，根据相对地址映照表的内容，加上正文段、ppda实际起始地址来设置系统中2张用户态空间页表，完成对新上台进程用户态空间的映射。

需要注意的是，MapToPageTable()函数对页目录第0项与第1项指向的进程页表进行装填，这并不会修改相对地址映照表的内容，所以在进程切换下台时不需要将页目录0、1项指向的进程页表保存至相对地址映照表。