第 12 章　内核内存管理

12.1　简　　介

操作系统管理所有的物理内存，并为内核中的不同子系统以及用户进程分配内存。当系统启动时，内核为其代码和静态数据结构保留部分物理内存，这部分内存将不再释放，并且不能再挪为它用①。内核对其余的内存空间进行动态管理——不同的客户程序(如用户进程，内核子系统)通过内核动态分配内存，并在不再使用它们时将它们释放掉。

UNIX 将内存划分为固定尺寸的片或页面。页面的大小均为 2 的整次幂，通常为 4KB②。UNIX 采用了虚存系统，进程空间中连续的页在物理内存中不必是连续的。本章之后的 3 章将介绍虚存系统。内存管理子系统维护进程的逻辑(虚)页与实际的物理内存之间的映射。当请求一个连续的内存块时，内存管理子系统可以通过分配若干物理上不连续的页面来满足这个请求。

虚存系统大大简化了页面分配的工作。内核为空闲页面维护一个链表结构。当进程需要页面时，内核从空闲链表中去掉足够的页面；当释放页面时，内核再将它们链入链表结构中。页的实际物理位置是无关紧要的。4.3BSD 中的 memall()和 memfree()，以及 SVR4 中的 get_page()和 freepage()就是这种所谓的页面级分配器的一种实现。

页面级分配器有两个主要的客户(图 12-1)。一个是虚存系统的分页系统，它为用户进程分配页面。在许多 UNIX 系统中，分页系统还为磁盘缓冲提供页面。另一个客户是内核内存分配器，它为不同的内核子系统提供各种尺寸的内存块，通常这些内存块仅使用很短的一段时间。

图 12-1　内核中的内存分配器

下面列出了一些使用内核内存分配器的模块：

- 路径名解析例程。它需要分配缓冲区来拷贝用户空间的路径名。
- allocb()例程需要分配任意大小的 STREAMS 缓冲区。
- 在许多的 UNIX 实现中，要为结束的进程分配 zombie 结构，保存进程的返回值和资源使用信息。
- 在 SVR4 中，内核需要动态分配各种对象(诸如 proc 结构，V 节点，文件描述块等等)。

大多数分配请求所要求的内存大小都远小于一个页面的大小，很明显不能用页面级分配器来实现这些功能。我们需要一套独立的机制来实现更细粒度上的内存分配。一个简单的解决方案就是避免动态地进行内存分配，早期的实现的 UNIX[Bach 86]为 V 节点、proc 结构等数据结构分配固定大小的内存。当系统需要临时保存路径名或网络报文时，系统借用磁盘块缓冲系统中的缓冲区。此外，针对特殊情况，还设计了一些 ad hoc 的分配机制，如终端驱动程序中的 clists 结构。

这种策略有许多弊端。首先，它极不灵活。各种表或缓冲区的大小在系统启动或编译时就被固定下来，不能根据系统的请求而调整。系统开放人员根据典型系统的工作负荷来选择表格的缺省大小，尽管系统管理人员通常可以调节这些尺寸，但这样做缺少指导依据。如果某些表过于小，这些表就很可能溢出，并使整个系统在没有任何征兆的情况下崩溃。如果相对保守地配置系统，所有的表都尽可能地大，则会造成内存的浪费。留给应用程序的内存相对减少。这也会造成系统的整体性能下降。

显而易见，内核需要一种通用的内存分配器，可以有效地处理不同尺寸的分配请求。在接下来的章节中，我们首先对这类分配器的功能和评测标准进行描述。之后，我们针对现代

UNIX 系统使用的不同的内存分配器进行介绍和分析。

12.2 功能需求

内核内存分配器(以下简称 KMA)处理动态内存分配请求。这些请求是由诸如路径名分析程序，STREAMS 系统，进程间通信系统等内核子系统发出的，KMA 不处理来自用户进程页的请求，这些请求由分页系统处理。

当系统启动后，内核为自身的代码和静态数据结构以及一些预先定义好的块缓冲池，如磁盘块缓冲等，保留空间。页面级分配器管理其余的物理内存，并以此来满足 KMA 和用户进程的动态分配请求。

页面级分配器预先为 KMA 分配一部分空间，KMA 要高效地使用这部分内存。有些实现不允许改变分配给 KMA 的内存容量，而有些实现则允许 KMA 从分页系统中借用页面，更有甚者，允许两个子系统间可以双向借用页面，分页系统也可以从 KMA 中借用空闲页面。

KMA 耗尽内存后，调用者被阻塞住，直至有足够的空闲内存后才恢复执行。调用者也可以在请求中携带一个标志，通知 KMA 在没有足够内存时返回一个失败返回值(通常为 NULL)而不是阻塞住调用者。这种选项对于中断处理程序是非常有用的。在分配失败后，中断处理程序需要做出相应的处理动作。例如，如果网络中断处理程序不能为接收到的报文分配内存，它可能简单地将报文丢掉，并等待发送者重发。

KMA 必须监视内存池中哪部分是正在使用的，哪部分是空闲的。内存被释放后，它应该可以用来满足其他分配请求。在理想情况下，当且仅当内存真正满时，即全部空闲内存的大小小于请求的大小时，分配内存请求才失败，而实际上，内存分配失败要比理想情况发生的快得多。这主要是碎块的问题，尽管有足够的内存，但它们并不是连续的一块，无法满足分配请求。

12.2.1 评估标准

高效地使用内存，尽可能减少内存浪费是评估内存分配器的一个很重要的标准。物理内存是有限的，分配器必须有效地使用这些内存空间。评价这方面的一个参数就是内存使用率，即总的内存分配请求的大小与满足这些请求所使用的内存大小池。理想的内存分配器的使用率应是 100%，而实际上，50%的使用率就是可以接受的了[Korn 85]。造成内存浪费的主要因素是碎块。空闲内存被分成若干小块，以致于小的不能再用了。分配器可以将两块毗邻的空闲块合并起来的方法减少碎块。

KMA 也必须是快速的。有许多内核子系统使用 KMA，这里面当然包括中断处理程序，它的性能是非常重要的。平均延迟和最坏延迟都是很重要的参数。用户进程可以在堆栈中简单地分配数据对象，而内核栈一般都很小，内核在很多情况下必须动态分配内存。这使得分配内存的速度尤为重要。速度较慢的分配器会导致整个系统的性能下降，

分配器还须有一个简洁的编程接口，适应小同类型的客户。一种可能的接口就是使用与标准 C 语言库用户级内存管理中的 malloc()和 free()相仿的接口。

```
void * malloc(size_t nbytes);
void free(void * ptr);
```

该接口一个显著的优点就是 free()例程不需要知道释放内存的大小。通常一个内核函数分配一块内存，并将它传给其他子系统，最后由该子系统释放这片内存。例如，一个网络驱动程序为接收到的报文分配内存，并将它传送给更高层的模块，由它们处理数据并释放内存。释放内存的模块可能根本就不知道分配内存的大小。如果 KMA 可以维护这些信息，将大大简化其客户的工作。

对接口的另一个要求是不强制要求客户一次性释放释放所有分配的内存。如果客户仅仅释放了部分内存，分配器应该正确地处理这种情况。但 malloc()/free()接口不允许这样的

情况。free()例程将释放整个空间，或者因地址与 malloc()分配得到的地址不匹配而失败。允许客户改变分配空间的大小(例如 realloc())也是很有用的。

分配的内存应该是对齐的，这样可以加速存取的速度。在许多的 RISC 体系结构中对齐是必须的。对于大多数系统而言，长字对齐就可以了，但在像 DEC 的 Alph AXP[DEC 92] 那样的 64 位机器上，需要在 8 个字节上对齐。与此相关的一个问题是分配内存的最小单位，通常是 8 或 16 个字节。

许多商用环境都具有周期性的使用模式。比如，某台机器白天用来进行数据库查询和事务处理，而到了晚上则进行数据库备份和重构。这些活动有着不同的内存需求。事务处理可能需要若干小的内存块来实现数据库加锁，而备份则要求分配给用户进程尽可能多的内存。

许多分配器将资源划分为独立的区，或桶，这些区对应不同尺寸的请求。比如，一个桶保存所有 16 字节的内存块，而另一个则保存 64 字节的内存块。这种分配器必须小心处理突发性和上面所说的那种周期性的内存使用模式。在某些分配器中，一旦内存分配到某个桶中，它就再不能用于满足其他尺寸的分配请求了，从而导致某些桶中有大量的内存，而其他的桶中内存不足。一个好的分配器应该提供一种机制，动态地回收某些桶中过剩的内存。

最后，与分页系统的交互作用也是一个重要的标准。但 KMA 中最初分配的内存用光后，它应该可以从分页系统中借用页面。分页系统也应该可以从 KMA 中回收过剩的空闲内存。这种交互应该是可以被高效地控制，确保公平性，避免任何一个系统的饥饿现象。

我们现在来看看集中分配方法，并用上面的方法对它们进行分析。

12.3 资源映射图分配器

资源映射图是一系列记录空闲内存的偶对<base，size>的集合(见图 12-2)。一开始，资源映射图只有一个项，它的 base 就是内存池的基址，而 size 就是内存池的大小(见图 12-2(a))。随着客户不断地分配和释放内存，内存池被逐渐打碎。内核为每一段连续的空闲内存分配一项。资源映射图按基址顺序组织，这样就可以很容易地连接两个邻接的空闲内存区。

利用资源映射图，内核可以用如下三种策略之一来满足分配请求：

·最先匹配　从第一个可以满足请求的空闲块中分配内存。这是最快的算法，但不利于减少碎片。

·最佳匹配　选择可满足请求的空闲块中最小的一个分配内存。该策略的缺点是可能会造成若干非常小的，不可用的空闲块。

·最差匹配　如果没有正合适的空闲块，选择可满足请求的空闲块中最大的一个。这看上去似乎不合常理，但它是处于这样的考虑：保证分配后剩下的空间足够大，以便满足后续的请求。

没有一种算法可以很好的匹配所有的使用模式。[Knut 73]中对这些算法和其他一些算法进行了详细的分析。UNIX 使用最先匹配策略。

图 12-2 给出了一个管理 1024 字节大小内存的资源映射图。它支持两个操作：

```
offset_t rmalloc(size);    /* returns offset of allocated region */
void rmfree(base, size);
```

最初(图 12-2(a))，整个区域都是空闲的，用一项来描述。接着来了两个请求，分别分配 256 和 320 字节。这之后有一个在偏移 256 处释放 128 字节的请求。图 12-2(b)给出了这些操作后资源映射图的状态。它有两个空闲区，用两项来描述。

接下来，在偏移 128 处释放 128 字节。分配器发现该区域与偏移 256 处的空闲块邻接。分配器将它们合并为一个 256 字节的区域，如图 12-2(c)所示。最后，图 12-2(d)给出了又发生了许多操作后资源映射图的状态。注意，尽管总的空闲区大小是 256 字节，但分配器已不

能满足任何大于 128 字节的请求了。

图 12-2　使用资源图分配器

12.3.1　分析

资源映射图是一种简单的分配器，它主要有如下优点：

·算法易于实现。

·资源映射图并非仅能用于内存分配。它能管理任何有序的，需要分配和释放连续块的对象(如页表项，以及以后将讲述的信号灯等)。

·它可精确地分配请求所需的内存，不浪费空间。通常在实际使用中，为了简单和对齐的要求先将请求取整到 4 或 8 的倍数。

·并不强制客户精确地释放分配的内存。如前面的例子可知，客户可以释放部分内存，由于 rmfree()的参数中携带了尺寸这个参数，分配器能够正确地处理这种情况。而且，空闲资源信息是与被分配的内存分离维护的。

·分配器完成邻接空闲块的合并，有利于内存的再利用。

然而，资源映射图分配器同时存在一些显著的缺点：

·在分配器运行一段时间后，资源映射图碎片化非常严重，生成许多很小的空闲块。这将使利用率显著下降。特别是此时分配器已无法满足较大的内存分配请求了。

·随着分片增加，资源图的大小也在增加，为每一片内存区建立一个表项。如果资源图预配置成固定数目的表项，它就有可能溢出，分配器就不能再记录某些空闲内存区了。

·如果资源图动态的增长，它需要另外一个分配器来为自己分配内存。这成了递归问题了，我们将在后面给出一个解决方案。

·为了合并邻接的空闲块，分配器必须按基址升序排列它们。排序的代价很高。在某些情况下，如资源映射图采用固定尺寸的数组实现时，情况会更糟。既便资源映射图动态分配并组织成链表，排序仍然是十分耗时的。

·分配器为找到足够大的空闲块必须对资源映射图进行线性搜索。随着碎片的增加，极端情况下搜索耗时很大，变得非常慢。

·尽管可以将内存池尾部的空闲页面返回给分页系统，算法并没有这样设计。事实上，分配器从来不减少内存池的大小。

资源映射图算法的性能很差。这一点使它不适合成为一个通用内核内存分配器。但它还是适用于某些内核子系统的。System V 的进程间通信就使用资源映射图算法管理信号量集合和消息的数据区。4.3BSD 的虚存子系统也使用该算法管理映射用户空间的页表(请参看 13.4.2 小节)。

在某些情形下，映射图的管理可以得到改进。通常可以将映射图的每一项存放在空闲块的前几个字节中，这种方法不需要额外的内存，也不需要动态分配映射图。只需一个全局指向第一个空闲块的指针，每一个空闲块保存它自身的大小和下一个空闲块的指针。这种方法要求空闲块至少有两个字的大小(一个放尺寸，一个存指针)。一般通过强制分配和释放就可以满足这一点。在 9.5 节中描述的伯克利快速文件系统(FFS)就使用该方法的变体来管理目录块里的空闲空间。

尽管这一改进很适合于通用内存分配器，但对于像信号量集，页表项这样的没有空间来存放管理信息的对象来说，该方法还是不适用的。

12.4　简单 2 次幂空闲表

在用户级 C 语言库中经常使用 2 次幂空闲表算法实现 malloc()和 free()。这种方法使用一组空闲表，每一张表存放某一特定尺寸的空闲块。空闲块的大小均为 2 的整次幂。例如，图 12-3 中有 6 个空闲表，分别存放尺寸为 32，64，128，256，512 和 1024 的空闲块。

每一个空闲有一个字大小的头结构，可用空间不包括这个字。当内存块为空闲时，这个字存放下一个空闲块的指针，当内存块被分配后，存放其所在的空闲表的指针。有时这个字也可以存放内存块的大小，这有助于捕捉某些 BUG，但需要 free()例程通过该值计算出空闲表的位置。

客户调用 malloc()分配内存，其中一参数为请求的大小。分配器根据请求的大小选择尺寸最小的可以满足该请求的空闲表进行分配。这一计算过程附加额外的头结构的大小，并将得到的结果取整为 2 次幂。32 字节的内存块可以满足 0～28 字节的请求，64 字节的内存块可以满足 29～60 字节的请求，等等。分配器将空闲块从相应的空闲表中删除，并在头结构中记录空闲表的指针，然后将指向头结构后第一个字节的指针作为返回值返回。

图 12-3　2 次幂空闲表分配器

当用户释放内存时使用由 malloc()返回的指针作为参数调用 free()例程。用户不必指定释放的内存的大小，但是必须释放所有由 malloc()分配的内存。用户无法仅释放部分分配的内存。free()将指针减去 4 得到头结构的指针，再从头结构中获得空闲表的指针，并将该内存块加入空闲表中。

分配器可以预先为每个空闲表分配一定数量的内存块，也可以先让它们为空，然后在需要时调用页面级分配器动态生成那些内存块。因此，如果某个空闲表为空，分配器处理相应尺寸的 malloc()请求时有 3 种可能：

- 阻塞请求，直至有相应尺寸的内存块释放。
- 用更大的不为空的空闲表的内存块满足请求。
- 向页面级分配器申分页面，并创建相应尺寸的内存块。

每一种方法都各有长短，要根据实际情况来选择合适的方法。例如，内核可以在分配请求中附加一个优先级参数来提供选择的依据。这样，分配器阻塞不能满足的低优先级的请求，用另外两种方法之一满足高优先级的请求。

12.4.1　分析

上述的算法简单而且应该是快速的。它的最吸引人的地方就是避免了资源映射图算法中漫长的线性搜索过程，彻底去除了碎片问题。在存在可用的内存块时，算法的最坏情况性能是有界的。分配器还提供了一套熟悉的编程接口，而其一个很大的优点就是 free()例程不再需要内存块的大小这个参数。这样一来，一块分配好的内存可以传递给其他函数或子系统，并最终由它们利用内存的基址释放这片内存。但另一方面，该接口不允许仅释放部分分配的内存。

该算法还存在许多缺点。将请求的大小取整到 2 次幂很浪费内存，内存利用率很低，而且在内存块中存放头结构使得这个问题更为严重。许多内存请求大小都是 2 的整次幂的，对于这样的请求，浪费率达到 100%。请求大小 2n 加上头后，分配大内存大小必须为 2(n+1)。例如，一个 512 字节的请求，需要分配 1024 个字节约内存块。

该算法也不可能将相邻的空闲块拼接起来以满足大内存块的分配请求。一般每个内存块的大小一直都是固定的。唯一可以变通的地方是，有时可以用大的内存块来满足小内存的分配请求。尽管在某些实现中可以从分页系统中借用内存，但算法不会将额外的空闲页面返回给页面级分配器。

虽然该算法比资源映射图算法要快的多，但还可以进-步改进。比如，例 12-1 中对分配尺寸的取整循环很慢而且低效。

实例 12-1　Malloc()的初始实现

```
  void * malloc(size)
  {
    int ndx = 0;    /* free list index */
```

```
    int bufsize = 1<<MINPOWER; /* size of smallest buffer */
    size += 4;      /* account of header */
    assert(size<=MAXBUFSIZE);
    while(bufsize<size){
        ndx++;
        bufsize <<= 1;
    }
  ... /* at this point, ndx is the index of the appropriate free list */
  }
```

下一节将针对这些问题介绍一个改进算法。

12.5 McKusick-Karels 分配器

Kirk McKusick 和 Michael Karels 共同提出一个改进的 2 次幂分配器[McKu 88]，许多 UNIX 变体都使用了该分配器，如，4.4BSD，Digital UNIX 等，该改进算法去除了某些情况下的空间浪费，如分配请求恰好是 2 的整次幂的情况。它还优化了取整的计算，并且对编译时已知的尺寸省去了计算时间。

McKusick-Karels 算法需要将被管理的内存组织成一组连续的页面，并且同一页面内的所有内存块的大小是一样的(都是 2 的整次幂)。此外,它还用一个页面使用数组(kmemsizes[])管理页面，每一个页面可以是如下三种状态之一：

- 空闲　相应的 kmemsizes[]中的元素是一个指向下一个空闲页面元素的指针。
- 被划分为特定大小的内存块　kmemsizes[]是内存块大小。
- 跨页面内存块的一部分　该内存块首页相应的 kmemsizes[]元素保存内存块大小。

图 12-4 是一个尺寸为 1024 字节的例子。freelistarr[]是所有小于一个页面的空闲块链表的队头数组。

因为在同一个页面上的内存块大小相同，分配的内存块无需头结构来保存指向空闲链表的指针。free()例程将内存块地址低位掩掉获得页面基址，在通过对应的 kmemsizes[]元素找到内存块的尺寸。内存块头结构的去除避免了内存浪费，尤其是那些恰好为 2 的整次幂的分配请求。

对 malloc()的调用用一个宏来替代。该宏对请求的尺寸进行取整(分配的内存块无头信息)，并从相应的空闲链表中删除它。如果空闲链表为空，宏就调用 malloc()函数为请求分配一个或多个页面，并分割成相应大小的内存块。在该宏中，取整循环由一组条件表达式代替。

例 12-2 是一个针对图 12-4 中的内存池的实现。

实例 12-2 McKusick-Karels 分配器

```
#define NDX(size)    \
    (size)>128    \
    ?(size)>256?4:3    \
    :(size)>64    \
        ?2    \
        :(size)>32?1:0
#define MALLOC(space, cast, size, flags)    \
{    \
    register struct treelisthdr * tlh;    \
    if(size <= 512 && \
```

```
        (flh = freelistarr[NDX(size)])!=NULL)    {  \
            space = (cast)flh->next;     \
            flh->next= *(caddr_t *)space;  \
        }else  \
            space =(cast)malloc(size,flags);  \
    }
```

使用宏的主要优点是当分配大小在编译时已知的情况下，NDX()宏被编译器转换为常量，避免了原来的许多指令。还有一个宏处理内存块释放，只在大内存块等少数情况下才调用 free()函数。

12.5.1 分析

McKusick-Karels 算法比 12.4 节中给出的简单 2 次幂分配器有个明显的改进。它更快，占内存更少，对大内存块请求和小内存块请求同样有效。然而，算法仍有许多 2 次幂算法中固有的缺点。不能将内存从一个空闲链表移至另一个空闲链表，这使得分配器在突发使用模式下很不稳定。这种模式会在短时期内消耗大量的某一大小的内存块。并且，算法仍无法向分页系统返回页面。

12.6 伙伴系统

伙伴系统[Pete 77]是一种将空同块合并技术融入 2 次幂算法的分配方法③。它的基本方法是通过不断地对分大内存块来获得小内存块，并尽可能地合并空闲块。当一个内存块被对分后，每一部分称为另一部分的伙伴。

为了解释这一算法，让我们先考虑一个简单的例子(图 12-5)。这里，伙伴系统管理 1024 字节的内存，最小的分配请求是 32 个字节。分配器使用一张位图来维护内存中每一个 32 字节块的使用情况。如果某位置 1，则相应内存块正在使用。伙伴系统还为每一种可能大小的内存块(32 到 512 间的 2 的整次幂)维护一个空闽链表。最初，整个内存就是一个内存块，让我们考虑一下分配器接受如下请求序列并采取相应动作后的结果：

图 12-5 伙伴系统

1.分配(256)： 将内存块对分为两个伙伴——A 或 A'。将 A'加入 512 字节的空闲链表中。然后将 A 分为 B 和 B'，将 B'加入 256 字节的空闲链表中。将 B 返回给客户。

2.分配(128)： 发现 128 字节的空闲链表为空，检查 256 字节空闲链表，删除 B'。将 B'分为 C 和 C'，将 C'加入 128 空闲链表，返回 C。

3.分配(64)： 发现 64 字节空闲链表为空，从 128 字节空闹链表中删除 C'。将 C'分为 D 和 D'，将 D'加入 64 字节空闲链表中，并返回 D。图 12-5 就是此时的情形。

4.分配(128)： 发现 128 字节空闲链表、256 字节空闲链表均空，检查 512 字节空闲链表，删除 A'。将 A 分为 E 和 E'，再进一步将 E 分为 F 和 F'，E'加入 256 字节空闲链表，F'加入 128 字节空闲链表，返回 F。

5.释放(C，128)： 将 C'加入 128 字节空闲链表。此时如图 12-6 所示。

图 12-6 伙伴系统，阶段 2

到此为止，尚没有合并发生。假设下一个操作是：

6.释放(D，64)： 分配器注意到 D'也是空闲的，随即将 D 与 D，合并为 C'，然后又发现 C 也是空闲的，也是再次合并得到 B'。最后，将 B'加入 256 字节空闲链表，结果如图 12-7 所示。

有几点需要先说明一下：

- 请求必须先被取整为 2 的整次幂。
- 对于例中的每一个请求，对应的空闲链表都是空的。大多数情况下不是这样的。如果对应

的空闲链表有可用的内存块，分配器将直接使用它们；为空时才进行对分。
·内存块的基址和尺寸是定位其伙伴的全部信息。这是由于算法使每个内存块都是对于其尺寸是对齐的。比如，128 字节，基址为 256 的内存块，其伙伴的基址为 384，而一个 256 字节的基址为 256 的内存块，其伙伴基址为 0。
·每一个请求同时更新位图，反映内存块的新状态。在合并时，分配器通过位图获知内存块的伙伴是否空闲。
·虽然上面的例子使用一个 1024 字节的页面，分配器也可以同时管理几个不相连的页面，空闲链表集保存所有页面的空闲块。由于伙伴是通过页内偏移决定的，合并不受影响，但分配器必须为每一个页面单独维护一张位图。

12.6.1 分析

伙伴系统非常适合于合并空闲块。它提供了非常好的灵活性，使内存可以以不同的尺寸再利用。此外，与分页系统的内存交换也很便利。无论何时分配器需要内存，它就可以从分页系统中分配一个新页面并进行适当的对分。当释放例程合并出整个页面后，就可以返回给分页系统了。

该算法的最大缺点是它的性能。每释放一次内存块，分配器都尽可能地进行合并。当分配与释放交替进行时，算法可能刚合并完内存块，马上就又把它分裂了。合并的过程是递归进行的,这导致最坏情况下极差的性能,下一节我们将介绍 SVR4 是如何克服这些性能瓶颈的。另一个缺点是编程接口。释放例程需要内存块的基址和大小。此外，分配器只能释放整个内存块。由于内存块的一部分是没有伙伴的，释放部分内存是不可能的。

12.7 SVR4 Lazy 伙伴算法

简单伙伴系统的最大问题是不断进行的分裂和合并造成的性能不佳。正常情况下，内存分配器处于平稳状态。此时，正在使用的内存块数目始终保持在一个很小的范围内。在这样的情况下，合并不会带来什么好处，相反却浪费了很多时间。合并仅当突发情况下才是必须的。在突发情况下，内存块的使用模式有很大的临时性的变化。

定义合并延时为将伙伴合并为一个内存块或判断伙伴是否为空闲的时间。合并可以产生一个更大尺寸的内存块，并且递归进行，直至发现其伙伴不为空闲。在伙伴系统中，每次释放操作至少引起一个合并延时，一般情况下会更多。

一个直接的方法就是延缓合并的发生，直到有必要合并时才尽可能地合并内存块，虽然这样可以减少分配和释放的平均时间，但少数几个引起合并的释放操作变得非常慢。像中断处理函数这样的。时间临界函数可能会调用分配器，有必要对最坏情况下的分配器的行为进行适当的控制。我们希望有一条中庸之道，既延缓合并，又不等到非做不可的时候才去做，必须将合并的代价均匀地分散到若干请求中。[Lee 89]提出了内存块类的概念，并基于其上的高低水位线对合并进行控制。下面介绍的 SVR4 方法[Bark 89]采用了相似的思想，并且更高效。

12.7.1 Lazy 合并

释放内存块要经过两个步骤，首先，内存块加入空闲链表中，以便其他请求可以再次使用它。然后，在位图中标记内存块为空闲，并且有可能的话将其与其伙伴合并。这就是合并操作。正常的伙伴系统在每次释放时都进行这两个步骤。

Lazy 伙伴系统执行步骤 1，标记内存块为局域空闲(可以在该类中分配，但不能合并)。是否执行步骤 2 决定于该类的状态。在任何时刻，一类内存块共有 N 个内存块，其中有 A 个内存块是活动的，L 个局域空闲的，G 个全局空闲的(在位图中标记为空闲的，可以合并的)。因此：

$$N = A + L + G$$

根据这些参数值的不同，一个内存块类有如下 3 个状态：
· lazy　内存块的消耗处于平稳状态(分配和释放是均衡的)，没有必要进行合并。
· reclaiming　消耗进入临界状态，需要进行合并。
· accelerated　消耗进入不稳定状态，分配器必须更快地进行合并。
决定内存块类状态的关键参数称为 slack，定义为
slack = N - 2L - G

系统在 lazy 状态时，slack 大于等于 2；在 reclaiming 状态时等于 1；accelerated 状态时为 0。算法保证 slack 不会变为负值。[Bark 89]给出了为什么 slack 可以有效地反映内存块类状态的详实的证据。

当释放一个内存块时，SVR4 分配器将其加入空闲链表并检验内存块类的状态。如果处于 lazy 状态，分配器什么也不做，不在位图中标记内存块为空闲。这样的内存块称为延迟内存块，在其头结构中也做相应标记(只有空闲块才有头结构)。尽管它可以用来满足相同大小的分配请求，但他不能与其伙伴合并。

如果内存块类处于 reclaiming 状态，分配器在位图中标记内存块为空闲，并进行可能的合并。如果处于 accelerated 状态，分配器合并 2 个内存块，一个是刚释放的那个内存块，如果还有其他延迟内存块的话，另一个就是其中之一。分配器将合并后的内存块释放给下一级尺寸的空闲链表，然后再检查新内存块类的状态，并决定是否继续合并。所有这些操作都修改 slack 的值，重新计算该参数。

为了有效地实现算法，空闲链表组织成双向链的结构。延迟内存块放在队头，非延迟内存块放在队尾。这样，延迟内存块优先被分配，这正是我们所希望的(分配它们不需要更新位图，代价小)。此外，在 accelerated 状态下，另一个延迟内存块可以很快地从队头查到。如果队头是非延迟内存块，则说明没有延迟内存块。

相对于基本伙伴系统而言，Lazy 伙伴系统有一些实质性的改进。在稳定状态下，所有的空闲链表均处于 lazy 状态，不花费任何时间进行合并或分裂。即使某个空闲链表进入 accelerated 状态，至多合并两个延迟内存块，至多比基本伙伴系统差 1 倍。

[Bark 89]分析了在不同模拟负荷情况下伙伴系统和 Lazy 伙伴系统算法的性能。其结果表明，Lazy 伙伴系统的平均延时比基本伙伴系统好 10%到 32%。但正如预期的那样，Lazy 伙伴系统的性能变化更大，最坏情况下的释放性能更差。

12.7.2　SVR4 实现细节

SVR4 使用两种类型的内存池即大的和小的。每一个小内存池都是 4096 字节的内存，分成若干 256 字节的内存块。前 2 个内存块用于维护给内存池的数据结构(如位图等)，其余的内存块可以用来分配或分裂。内存池的分配大小是 2 的整次幂，范围从 8～256。大内存池是 16KB 的内存，分配大小从 512 一 lGK。在稳定状态下两种类型的活动内存池有很多。

分配器以内存池的大小为单位与分页系统交换内存。当其需要内存时，就向页面级分配器申请大内存池或小内存池。当内存池的所有内存块均被合并后，分配器将内存池返回给分页系统。

大内存池因其最大内存块的大小与内存池大小相同，仅使用 Lazy 伙伴算法进行合并。对于小内存池，伙伴系统仅能合并到 256 字节。需要一个独立的函数收集内存池中的 256 字节内存池。这个操作是很消耗时间的，必须在后台进行。SVR4 中有一个称为 kmdaemon 的系统进程周期性地进行合并，并将合并后的内存池返回给页面级分配器。

12.8　Mach-OSF/1 的 Zone 分配器

在 Mach 和 OSF/1[Sciv 90]系统中均使用 Zone 分配器完成快速的内存分配，并在后台进行垃圾收集工作。每一种动态分配对象类(如 proc 结构，凭证，消息头等)都分配一个 zo-ne，

即该类的空闲对象池。即使两个类的对象具有相同的尺寸，每个类仍有自己的 zone。例如，port translations 和 port sets(见第 6 章)都是 104 个字节[DEC 93]，但它们都有自己的 zone。此外，还有一组为 2 的整次幂 zone，供那些不需要私有对象池的各种客户使用。

Zone 一开始都是通过页面级分配器生成的，需要时可以由页面级分配器获得更多的内存。任何一个页面只能用于一类 zone，也即，在同一物理页面上的所有对象都是同一个类的。每个 zone 的空闲对象通过链表维护，队头是 struct zone。这些队头结构本身又是一个称为 zone of zones 的类中分配而来的。zone of zones 的每一个对象就是一个 struct zone。

每个内核子系统都要通过函数：

zinit(size，max，alloc，name);

初始化它使用的 zone，其中 size 是对象的大小，max 是 zone 可能使用内存的最大字节数，alloc 是每次空闲链表为空时再分配的内存大小，name 是一个描述 zone 的字串。zinit()从 zone of zones 中分配一个 zone 结构，记录 size，max 和 alloc。其后 zinit()向页面分配器分配 alloc 字节，将其分成一系列 size 大小的对象，加入空闲链表。所有活动的 zone 由一个链表维护，由全局变量 first_zone 和 last_zone 说明(见图 12-8)。链中第一个元素是 zone of zones，其他 zone 都是从这里分配的，

分配和释放都是非常快速的，只需从空闲链表中删除或增加对象。如果某个分配请求发现空闲链表为空时，就从页面级分配器再分配 alloc 字节。如果使用的内存达到 max 后，以后的分配请求就失败了。

12.8.1 垃圾收集

很明显，这样的管理模式需要垃圾收集，否则，一阵突发请求将使可用的内存所剩无几。为了不导致最坏情况下某些操作的异常行为，垃圾收集在后台运行。分配器维护一个称为 zone page map 的数组，每一个元素对应分配给某个 zone 的页内。该映射图的每一个项有两个计数变量：

- in_free_list 记录该页面上空闲对象的个数。
- alloc_count 是页面上的对象总数。

图 12-8 zone 分配器

当 zone 从页面级分配器分配页面时，首先设置 alloc_count。某些对象的大小不能整除页面大小，这样的对象可能跨两个页面。这时，在两个页面的 alloc_count 都统计该对象。分配和释放时并不对 in_free_list 更新，每次垃圾收集启动时才重新计算。这又进一步降低了分配、释放请求的延时。

对换任务在每次运行时通过调用 zone_gc()例程启动垃圾收集过程。它对整个 zone 链遍历两遍。第一遍，扫描空闲链表，统计 in_free_list。第一遍扫描后，如果页面的 in_free_list 和 alloc_count 相等，也就是说页面上所有的对象均为空闲的，页面可以重新利用④。在第二遍扫描中，zone_gc()从空闲链表中删除这样的对象。最后，它调用 kmem_free()将页面返回给页面级分配器。

12.8.2 分析

Zone 分配器快速而且高效，它有简单的编程接口。通过：

obj = void * zalloc(struct zone * z);

分配对象，其中 z 是指向对象 zone 的指针。通过：

void zfree(struct zone * z, void * obj);

释放对象。客户必须释放整个对象，而且必须知道释放的对象属于哪个 zone。不允许释放对象的一部分。

Zone 分配器有个很有趣的特性。它使用自己为新创建的 zone(使用 zone of zones)分配

zone 结构，这造成系统在启动时的“鸡和蛋”的问题。内行管理需要 zone 来分配其私有数据，而 zone 子系统又需要内存管理子系统的页面级分配器。解决这一问题的方案是使用一小块静态配置的内存，通过它来创建 zone of zones 。

Zone 对象的大小都恰好是请求的大小，不存在 2 次幂方法中的空间浪费问题。垃圾收集可以将空闲页面返回给分页系统，供其他 zone 使用，这提供了一种内存复用的机制。

垃圾收集的效率是一个重要的问题。由于它在后台运行，不对某个分配或释放请求产生直接影响。垃圾收集算法非常慢，算法中有一个线性遍历。首先是所有空闲对象，然后是 zone 的所有页面。垃圾收集在结束以前不会释放 CPU 资源，这会影响系统的反应能力。

[Sciv 90]声称垃圾收集没有严重影响并行编译测试的性能。然而，至今也没有关于垃圾收集代价的公开数据。此外，垃圾收集对整个系统性能的影响也很难估计，垃圾收集算法复杂而又低效,其与 slab 分配器(见 12.10 节)相比而言,后者有更简单和快速的临界收集机制。而且最坏情况下的性能也比前者更好。

12.9 多处理器的分层分配器

在共享内存的多处理机系统中，内存分配又有额外一些问题。传统系统使用的空闲链表、分配位图等数据结构不是多处理机安全的，必须使用锁进行保护。这将造成大规模的并行系统中对锁的竞争，CPU 经常处于等待锁被释放的状态下。

在 Dynix 的 UNIX 实现中给出了一个方案。Dynix 运行在 Sequent S2000[McKe 93]上，使用一种分层分配机制，采用 SystemV 的编程接口。Sequent 多处理机主要用于大型在线事务处理。在这种负荷下，该分配器工作的很好。

图 12-9 给出了分配器的设计原理。最低层(per-CPU 层)的操作速度最快，而最高层(coalesce-to-page 层)则完成合并页面这样的耗时的操作。还有一个 coalesce-to-rmblock 层(未画出)，用于大块内存(4MB)的页面分配。

per-CPU 层为每一个处理器管理一组 2 次幂内存池。不同处理器的内存池是隔离的，不必加锁就可以直接操作。大多数情况下，分配和释放操作都是快速的，只访问本地空闲链表。

当 per-CPU 空闲链表为空时，它通过 global 层补充。global 层维护一个自己的 2 次幂内存池。同样，per-CPU cache 中过量的空闲块也可以返回给 global 空闲表。为了进一步优化，这两层间的空闲块的移动采用成组方式(target-sized group，如图 12-9 中每一次移动 2 个空闲块)。这避免了不必要的链表操作。

为了实现成组方式移动，per-CPU 层维护两个链表——main 和 aux，分配和释放操作主要使用 main 链表。当它为空时,将 aux 上的空闲块移到 main 上,而 aux 则由 global 层补充。相似地,main 链表中的过剩空闲块(超过 target)也返回给 aux,aux 上的空闲块返回给 global 层。这样一来，至多每 target 次移动才访问 global 层一次。target 是一个可以调节的参数，增加 target 就会减少对 global cache 的访问次数，但同时使 per-CPU 层保存了更多的空闲块， global 层维护一组全局的 2 次幂空闲链表。每一个链表都分成若干个 target 空闲块。偶而因为内存不足或 per-CPU cache 清空内存块，也可以传送非 target 空闲块。这些内存块保存在单独一个链表中。当其中空闲快满 target 后，再移回 global 层的空闲链表中。

图 12-9 多处理机的层次分配器

当 global 层的空闲块超过一个目标值后，过剩的空闲块返回给 coalesce-to-page 层。该层为每一个页面维护 -个链表(同一页面上的内存块大小相同)。该层将空闲块加入相应的队列中，累计页面的空闲块数。当某个页面上所有的内存块均被释放后，将页面返回给分页系统。与此相反，coalesce-to-page 层也可以从分页系统获得页面，生成新的内存块。

coalesce-to-page 层根据每个页面的空闲块数排序。分配时，首先从那些空闲块最少的页面中分配。而有较多空闲块的页面有更长的时间等待其上的内存块释放，增加了页面被返

回给分页系统的概率。这有利于提高合并的效率。

12.9.1 分析

Dynix 为共享内存多处理机系统提供了高效的内存分配算法。它支持标准的 SystemV 接口，允许分配器与分页系统间交换内存。per-CPU 层减少了共享锁的竞争，双空闲链表加速了 per-CPU 层与 global 层间的内存块交换。

将 Dynix 的合并策略与 Mach 的基于 zone 的分配器进行一下对比，可以发现许多有意思的地方。Mach 的算法采用了一种标记-清除的方法，每次都要线性扫描整个内存池。这种操作由于太消耗计算资源，而不得不由一个独立的后台任务完成。在 Dynix 中，每次内存块被释放到 coalesce-to-page 层后，对应页面的数据结构都相应更新。当页面上的内存块均释放后，页面返回给分页系统。所有这些都是前台操作，是释放操作的一部分。每个释放操作所增加的负荷很小，并没有造成最坏的性能。

[MeKe 93]的测试结果表明，在单 CPU 环境下，Dynix 算法比 McKusick-Karela 算法快 3～5 倍，在多处理器环境下更好(25 个处理器是，100-1000 倍)。但这些比较都是在最好情况下的，即仅从 per-CPU 层进行分配。这份报告没有给出更一般的结果。

12.10 Solaris 2.4 的 Slab 分配器

Solaris 2.4 使用的 slab 分配器探讨了许多本章先前介绍的分配器忽略的性能问题。因而，slab 分配器比其他分配器有更好的性能和内存利用率。其设计重点主要有 3 个方面——对象复用，硬件 cache 使用率和分配器 footprint。

12.10.1 对象复用

内核使用分配器创建各种各样的临时对象，如 i 节点，proc 结构，网络报文缓冲区等等。分配器对一个对象的操作顺序大致如下：

1.分配内存。

2.构造(初始化)对象。

2.使用对象。

4.析构对象。

5.释放内存。

内核对象通常比较复杂，包含诸如引用计数，链表头，互斥锁，条件变量等子对象。对象的构造过程对这些域赋于一个固定的初值。析构阶段也处理同样的域。在某些情况下，可能在释放内存之前，这些域又恢复到它们最初的初值。

例如，-个 v 节点包含其所在页面链表的队头。当 v 节点初始化时，链表是空的，在许多的 UNIX 实现[Bark 90]中，内核仅在 v 节点上的所有页面均从内存中清空后才释放 v 节点。因此，就在释放 v 节点之前(析构阶段)，它的链表又恢复到空状态。

如果内核对另一个 v 节点使用同一个对象时，就不必对链表队头重新初始化了，析构过程可以处理这一点。其他被初始化的域也有相同的原理。例如，内核分配一个对象，其初始化引用计数为 1。当所有引用释放后释放该对象。互斥锁其初始状态为解锁状态，而且必须解锁后才能释放它。

这些都是缓存并复用对象比分配内存块并初始化优越的地方。由于避免了必须是 2 的整次幂的条件，对象缓存还有效地利用了空间。zone 分配器(12.8 节)也同样基于对象缓存，有非常好的内存利用率，但由于它没有考虑对象的状态，没有消除重复初始化的代价。

12.10.2 硬件 Cache 利用率

传统的内存分配器都对使用硬件 cache 增加了一个潜在的问题。大多数处理器都有一个容量小且简单的一级数据 cache，其大小是 2 的整次幂(15.13 节有更详细的介绍)。MMU 根据如下计算公式将不同的地址映射到 cache 单元上：

cache location＝address%cache size;

硬件访问某处地址时，首先在 cache 单元中检查数据是否被缓存了。如果没有缓存，就从内存中获取，并重写 cache 单元的内容。

典型的内存分配器，如 McKusick-Karels 算法和伙伴算法，将请求内存尺寸取整为 2 的整次幂，其分配地址都是对于相应尺寸对齐的。此外，大部分内核对象都在对象的开始部分保存非常重要的，经常访问的域。

这两个因素的影响是非常显著的。比如，考虑这样-个实现，内存中的 i 节点大小是 300 字节。起始 48 个字节要经常访问。内核将其分配在一个相对 512 字节对齐，大小为 512 的内存块上。在这 512 个字节中，只有 48 个字节(8%)是经常使用的。

结果，硬件 cache 中接近 512 字节的边界处竞争激烈，而其余部分利用率很低。这种情况下，i 节点只利用了 9%的 cache。其他对象也有类似的情况。内存地址的不正常分布造成硬件 cache 不能有效地利用，并进而导致内存访问性能下降，

这种问题在采用多条总线的分体存储器系统中会更为严重。例如，在 SPARCcenter 2000[Cekl 92]上，以 256 字节为步长通过 2 条总线访问不同内存体。对于上面那个例子，大部分操作都访问总线 0，造成了总线使用的不均衡。

12.10.3 分配器 footprint

分配器的 footprint 是硬件 Cache 和快表(TLB)中被定位自身覆盖的部分(13.3.3 节详细介绍 TLB)。使用资源映射图或伙伴算法必须检查几个对象才能找到合适的内存块。这些对象分布于内存中不同的地方，经常彼此十分遥远。这时就引起 cache 或 TLB 的失效，降低了分配器的性能。分配器访问内存是将覆盖那些“热”(活动的)cache 行或 TLB 表项，访问它们需再次从内存中装载。这对性能的影响更大。

像 MeKusick-Karels 那样的分配器的 footprint 很小，分配器仅通过简单的计算确定对应的内存池，并简单地从空闲表中移走内存块。slab 分配器也使用同样的原理控制 footprint。

12.10.4 设计与接口

slab 分配器是 zone 分配器的变体，也有一组对象缓存。每个缓存保存一种对象类型，如 v 节点缓存，proc 结构缓存等等。正常情况下，内核从它们各自的缓存中分配和释放对象。分配器也可以向缓存补充内存和从缓存中返回过剩的内存。

每个缓存分为前端和后端两部分(图 12-10)。前端与客户直接交互。客户从缓存中获取构造好的对象，返回析构的对象。后端与页面级分配器交互。随着系统使用模式的变化，不断地与其交换页面。

图 12-10 slab 分配器的设计图

内核子系统通过调用：

cachep = kmem_cache_create(name, size, align, ctor, dtor);

初始化某类对象的缓存。其中，name 是描述对象的宇串，size 是对象的大小，以字节为单位，align 是对象需要的对齐参数，ctor 和 dtor 分别是构造函数和析构函数指针。该函数返回对象缓冲的指针。

内核分配对象时调用：

objp = kmem_cache_alloc(cachep, flags);

释放对象时调用：

kmem cache_free(cachep, objp);

由于内核在释放对象时已恢复了对象的初始状态，复用对象时不必进行构造和析构。如 12.10.1 节中所述，恢复状态都是自动完成的，无需额外的操作。

当缓存为空时，调用 kmem_cache_grow()从页面级分配器中获得 slab 内存，在其上创建

对象。slab 是由若干连续页面组成一整块内存，由 cache 进行管理。slab 可以容纳若干对象实体，缓存用 slab 中的很少一部分存放管理数据，将其余的 slab 分成若干对象大小的内存块。最后调用对象的构造函数(kmem_cache_create()参数中的 ctor)进行初始化，加入缓存。

当页面级分配器需要恢复内存时，调用 kmem_cache_reap()。该函数查找所有对象均为空的 slab，析构所有对象(调用 kmem_caehe_create()参数中的 dtor)，从缓存中删除它们。

12.10.5 实现

slab 分配器针对大对象和小对象使用不同的技术。我们先来看看小对象，每个小对象可以存放在一个页面上。页面分为三部分——kmem_slab 结构，对象集，不用的空间(见图 12-11)。kmem-slab 占 32 个字节，位于 slab 的末尾。每个对象使用额外的 4 个字节保存空闲列表的指针。slab 区中分配对象后剩下的区域是不使用空间。例如，如果 i 节点有 300 字节，一个 4096 字节的 slab 可容纳 13 个 i 节点，剩下 104 个字节无用(除去 kmem_slab 结构和 13 个空闲列表指针)。这部分不使用空间又分成两部分，下面将给出解释。

图 12-11 小对象的 slab 组织

kmem_slab 结构有一个使用中对象计数变量，还有链接用一 cache 其他 slab 的双链表指针，及空闲链表指针。每个 slab 维护自己的单链空闲链表。在紧接着每个对象的 4 个字节中保存空闲链表的链接数据。只有空闲对象需要这个域。为了避免覆盖对象的数据状态，必须将它从对象中区分出来。

不使用空间分为两部分：在 slab 前面的 slab 着色区，其余的部分放在 kmem_slab 结构的前面。分配器根据对齐的要求选用不同大小的着色区。在上面的那个 i 节点的例子中，如果 i 节点需要在字节 8 上对齐，slab 可以有 14 个不同的着色值(0～104，间隔 8)。这就使这类对象有良好的基址分布，从而带夹均衡和高效的硬件 cache 和存储总线使用率。

内核分配对象时从空闲队列中删除第一个元素，并累加 slab 的使用中计数变量。释放对象时，通过公式：

slab address = object address%slab size;

得到 slab 空闲链表的指针。将对象加入其中，将使用中计数变量减 1。

在使用中计数变为 0 时，整个 slab 就空闲了，slab 就可以回收了。缓存将所有 slab 组成一个有序的双向链表，完全活动的 slab(所有对象均在使用)排在前，部分活动的 slab 居中，空闲 slab 排在最后。缓存还有一个指向第一个有空闲块的 slab 的指针。在有分配请求时，从该 slab 上分配对象。因此，缓存并不是使用一个完全空闲的 slab 分配对象。直至所有部分活动的 slab 用完后，才使用空闲 slab。在页面级分配器回收内存时，它检查缓存的队尾，并删去空闲的 slab。

大对象 Slab

上面的实现对占用多个页面的大对象来说，并不能很有效地使用内存。对于这样的对象，缓存从一个独立的内存池(当然是另一个缓存)中分配 slab 的管理数据结构。除了 kmem_slab 结构之外，缓存还为每个节点分配了一个 kmem_bafctl 结构。该结构包括空闲链表指针，指向 kmem_slab 的指针，一个指向对象自己的指针。此外，slab 还维护一张 hash 表，完成从 object 到 kmem_bufctl 结构的反向转换。

12.10.6 分析

slab 分配器是一个设计优良，功能强大的工具。它有效地利用了空间，用于管理的数据结构仅有 kmem_slab 结构，一个不使用空间和每个对象的链接指针。大多数请求响应的时间都很短，slab 分配器只是简单的从空闲链表中删除对象并更新使用中计数变量。slab 分配器的着色机制提高了硬件 cache 和内存总线的利用率。这大大提高了系统的整体性能。山于 slab 分配器每次请求只访问一个 slab，因而它还有很小的 footprint。

垃圾收集算法比 zone 分配器要简单得多，但原理基本相同。每个请求都修改使用中计数

变量，垃圾收集的代价均匀地分给每一个请求。真正的回收操作因必须寻找一个空闲 slab 而扫描不同的缓存，这会带来一些额外代价。最坏情况下的性能与缓有的总数成正比，而不是 slab 的个数。

slab 分配器的一个缺点是要为每一类对象都维护一个缓存而带来的管理负担。对 于常用的对象类，其缓存大而且经常使用，这个代价并不明显。对于那些小的不经常使用的缓存，这个代价就是不可接受的了。该问题在 zone 分配器中也存在。可以为那些不必有自己的缓存的对象分配一组 2 次幂内存池来解决这个问题。

slab 分配器再增加了像 Dynix 那样的每个处理器的私有缓冲，其性能会有所改善。[Bonw 94]同意这一点，并说这会带来进一步的改进。

12.11 小　　结

通用内核内存分配器的设计涉及许多重要问题。它必须快速、易用，高效地使用内存。我们已经介绍了几个分配器，并分析了它们的优点和缺点。资源映射图分配器是唯一允许释放部分分配的内存，但其线性搜索的方法对大多数应用的性能不佳。McKusick-Karels 分配器有最简单的接口，使用标准的 malloc()和 free()语法，但不能合并内存块，也不能返回过剩的内存给页面级分配器。伙伴系统不断地将内存对分或合并，以此来适应内存请求的变化，但它的性能通常很差，特别当合并比较频繁的时候。zone 分配器通常很快，但其垃圾收集机制效率很低。

Dynix 和 slab 分配器比其他方法有显著的提高。Dynix 使用 2 次幂算法，但为每一个处理器提供了一个缓存，还有一个快速的垃圾收集算法(coalesce-to-page 层)。slab 分配器是 zone 算法的改进算法。它通过对象复用和均匀地址分布使性能得以改善。此外，它使用了一个简单的垃圾收集算法，使最坏情况的性能得到控制。正如前面提到的，为每个 CPU 增加缓存使 slab 分配器更好。

表 12-1 给出了一组 slab 分配器，SVR4 分配器和 McKusick-Karels 分配器比较的实验结果[Bonw 94]。数据表明，对象复用使分配和初始化时间依不同的对象减少了 1.3～5.1 倍。表中也列出了除改进的分配时间之外的其他改进。

表 12-1　几种分配器的性能

	SVR4	McKusick-Karels	slab
平均分配+释放时间(微秒)	9.4	4.1	3.8
总碎片(浪费)	46%	45%	14%
Kenbus 测试性能(每秒执行脚本的数目)	199	205	233

这些技术中的许多也可以用于用户级内存分配器。用户级分配器的需求与内核分配器相比很不一样，因此，一个好的内核分配器可以在用户级工作的很好或反之。用户级分配器有一个容量很大的(虚)内存，除了那些极耗内存的应用外，几乎是没有什么限制的。因此，对它而言，合并内存块，根据请求的变化而调整内存等并不太重要，相反快速的分配和释放才是重要的，接口可能出许多不同的，独立的应用使用，简单而标准的接口也是非常重要的。[Korn 85]还介绍了几种不同的用户级分配器。

12.12 练　　习

1. 内核内容分配器在哪些方面不同于用户级分配器?

2.对于一个有 n 个元素的资源,资源映射图的最大尺寸是多少(即资源映射图元素的个数)?

3.写一个程序，用一组模拟请求统计一下资源映射图分配器的内存利用率和性能。用该程序比较最先匹配，最佳匹配，最差匹配三种策略。

4.实现 McKusick-Karels 分配器的 free()函数。

5.写一个 scavenge()例程，合并 McKusick-Karels 分配器中的空闲页面，并将它们返回给用户级分配器。

6.实现一个管理 1024 字节内存区，最小分配尺寸是 16 字节的伙伴算法。

7.生成一个可以使伙伴系统产生最坏性能的请求序列。

8.在 12.7 节中描述的 SVR4 lazy 伙伴算法中，下列事件将使 N，A，L，G，slack 的值怎样变化?

(a)当 slack 大于 2 时释放一个内存块。

(b)分配一个延迟内存块。

(c)分配一个非延迟内存块。

(d)slack 等于 1 时释放一个内存块，但此时所有的空闲块田其伙伴尚不为空闲而不能合并。

(e)一个内存块与其伙伴合并。

9.哪个内存分配器可以在多处理器系统中增加 Dynix 中每个 CPU 的缓存?哪个算法不能使用这样的技术?为什么?

10.为什么 slab 分配器对小对象和大对象使用不同的实现方法?

1l.本章中介绍的分配器哪个具有简单的编程接口?

12.哪个算法允许客户释放部分分配的内存?

13.哪个分配器可能在内核还有足够大的内存块时拒绝分配请求?

①许多现代 UNIX 系统(如 AIX)可以对部分内核进行换页管理，

②这是一个软件定义的页尺寸，不一定要等于硬件页面大小。硬件页面大小是由存储管理单元决定的保护和地址转换的粒度。

③ 这是双伙伴系统，它是最简单和使用最广泛的伙伴系统。我们还可以将缓冲区分成 4 份，8 份或更多份来实现其他伙伴算法。

④如果被捕捉的页面在其上部或底部有对象跨越了两个页面，这些对象必须首先移到空闲队列上，并且要相应地递减另一个页面上的 alloc_count。