9.虚拟内存

9.1 物理和虚拟寻址

现代处理器使用的是一种称为虚拟寻址的寻址方式。

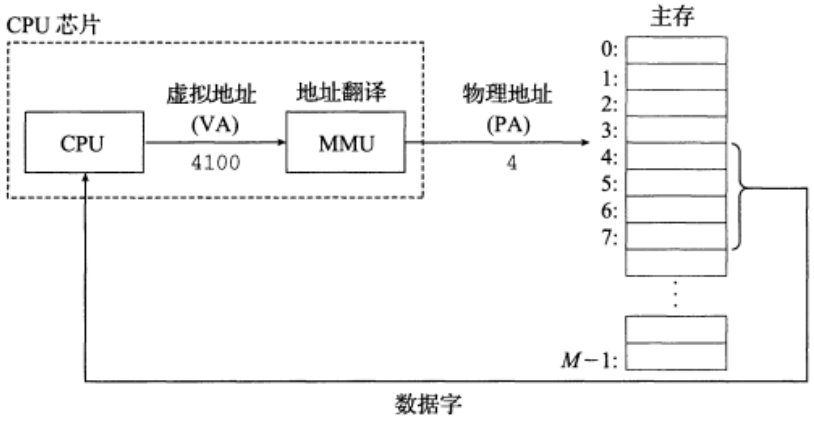


图 9-2　一个使用虚拟寻址的系统

9.2 地址空间

地址空间是一个非负整数地址的有序集合。
一个地址空间的大小是由表示最大地址需要的位数来描述的。例如，一个包含N=2n次方地址的虚拟地址空间就叫做一个n位地址空间。

9.3 虚拟内存作为缓存的工具

这一节中虚拟内存是所有的存储空间（包括磁盘等存储设备），会映射到主存（物理存储）空间上）
VM系统通过将虚拟内存分割为称为虚拟页的大小固定的块来处理磁盘和主存之间的传输单元。每个虚拟页的大小为P=2p次方字节。类似的，物理内存被分割为物理页，大小也为P字节（物理页也称为页帧）。
在任意时刻，虚拟页面的集合被分为三个不相交的子集：

- 未分配的：VM系统还未分配（或创建）的页。未分配的块没有任何数据和他们相关联，因此也就不占用任何磁盘空间。
- 缓存的：当前已缓存在物理内存中的已分配页。
- 未缓存的：为缓存在物理内存中的已分配页。

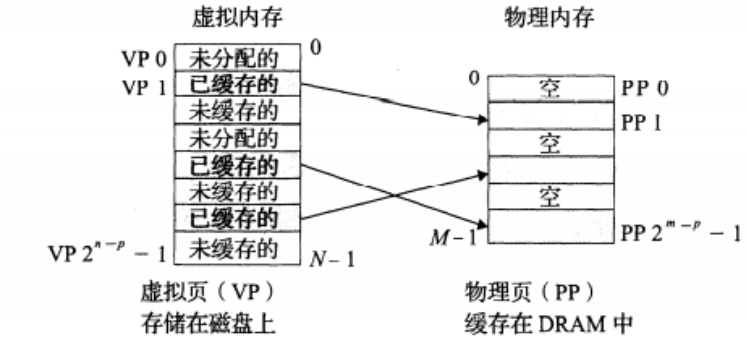


图 9-3　一个 VM 系统是如何使用主存作为缓存的

9.3.1 DRAM缓存的组织结构

因为大的不命中和访问第一个字节的开销，虚拟页往往很大。同时DRAM缓存是全相联的。因为访问磁盘时间长，DRAM缓存采用写回而不是直写。

9.3.2 页表

虚拟内存系统必须有某种方法来判断一个虚拟页是否缓存在DRAM中的某个地方，并且如何找到它。这些功能是由软硬件联合提供的，包括OS软件，MMU（内存管理单元）中的地址翻译硬件和一个存放在物理内存中叫做页表的数据结构，页表将虚拟页映射到物理页。

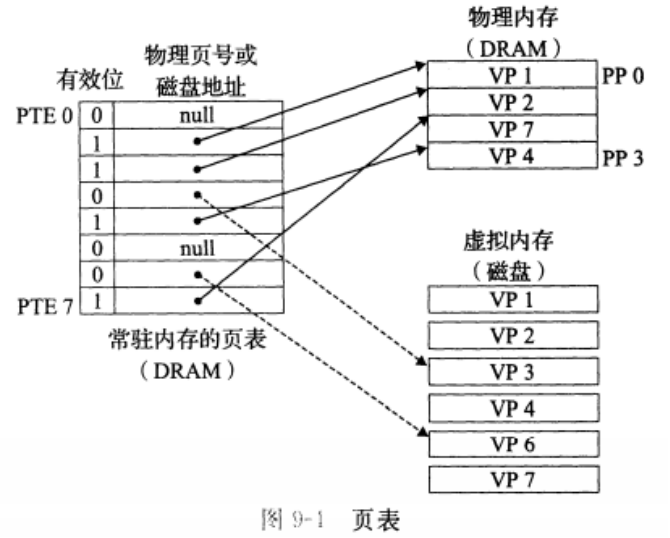


图 9-1 页表

这里的页表是虚拟内存表示全部内存时
页表就是一个页表条目（Page Table Entry，PTE）的数组。虚拟地址空间中的每个页在页表中一个固定偏移量处都有一个PTE。

有效位和地址：

- 有效位为1，地址字段表示了DRAM中相应的物理页的起始位置（已分配已缓存）
- 有效位为0，地址字段非空，地址字段则表示该虚拟页在磁盘中的位置（已分配未缓存）
- 有效位为0，地址字段空，未分配未缓存

9.3.3 页命中

9.3.4 缺页

9.3.5 分配页面

9.3.6 又是局部性救了我们

9.4 虚拟内存作为内存管理的工具

虚拟内存小于物理内存，也就是OS学的那种
OS为每一个进程提供了一个独立的页表，因而也就是一个独立的虚拟地址空间。多个虚拟页面可以映射到同一个共享物理页面上。

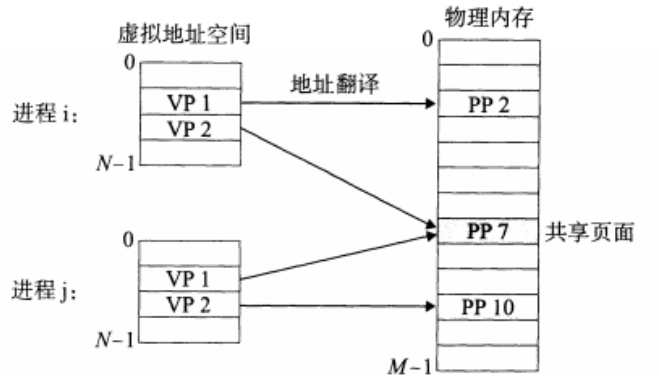


图 9-9 VM 如何为进程提供独立的地址空间。操作系统为系统中的每个进程都维护一个独立的页表

9.5 虚拟内存作为内存保护的工具

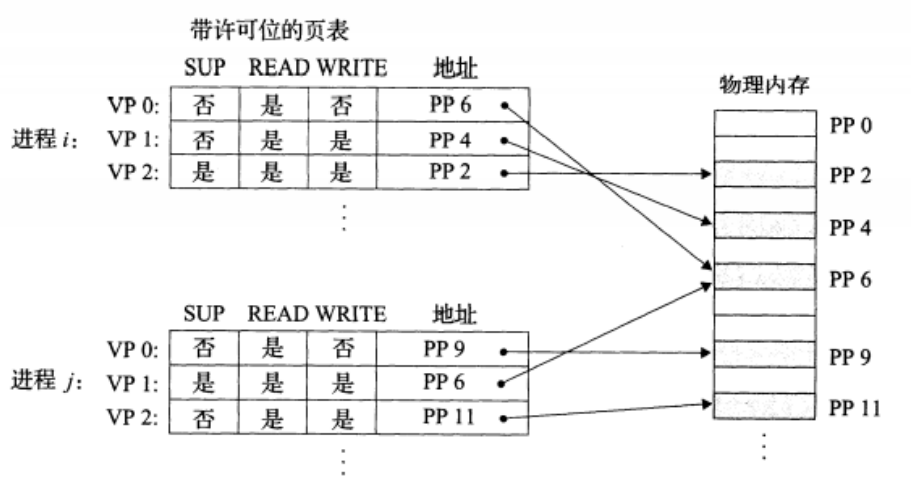


图 9-10 用虚拟内存来提供页面级的内存保护

如果一条指令违反了这些许可条件，CPU就触发一个一般保护故障，将控制传递给一个内核中的异常处理程序。

9.6 地址翻译

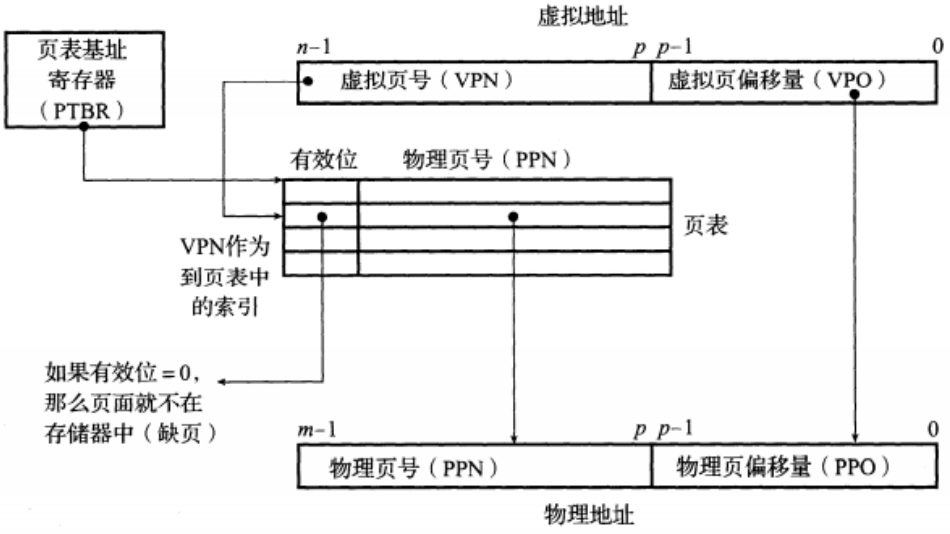


图 9-12 使用页表的地址翻译

图 9-12 展示了 MMU 如何利用页表来实现这种映射。CPU 中的一个控制寄存器，页表基址寄存器(Page Table Base Register，PTBR)指向当前页表。n 位的虚拟地址包含两个部分：一个 p 位的虚拟页面偏移(Virtual Page Offset，VPO)和一个($n-p$)位的虚拟页号(Virtual Page Number，VPN)。MMU 利用 VPN 来选择适当的 PTE。例如，VPN 0 选择 PTE 0，VPN 1 选择 PTE 1，以此类推。将页表条目中物理页号(Physical Page Number，PPN)和虚拟地址中的 VPO 串联起来，就得到相应的物理地址。注意，因为物理和虚拟页面都是 P 字节的，所以物理页面偏移(Physical Page Offset，PPO)和 VPO 是相同的。

图 9-13a 展示了当页面命中时，CPU 硬件执行的步骤。

- 第 1 步：处理器生成一个虚拟地址，并把它传送给 MMU。
- 第 2 步：MMU 生成 PTE 地址，并从高速缓存/主存请求得到它。
- 第 3 步：高速缓存/主存向 MMU 返回 PTE。
- 第 4 步：MMU 构造物理地址，并把它传送给高速缓存/主存。
- 第 5 步：高速缓存/主存返回所请求的数据字给处理器。

9.6.1 结合高速缓存和虚拟内存

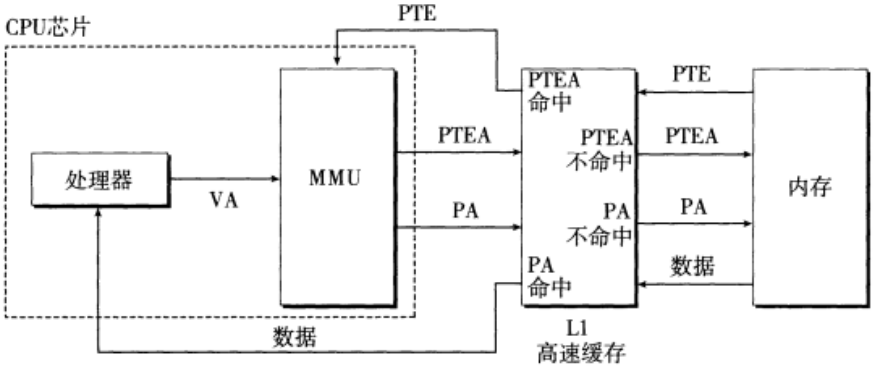


图 9-11 将 VM 与物理寻址的高速缓存结合起来(VA：虚拟地址。PTEA：页表条目地址。PTE：页表条目。PA：物理地址)

9.6.2 利用TLB加速地址翻译

在MMU中包括一个关于PTE的小的缓存，称为翻译后备缓冲器（Translation Lookaside Buffer，TLB)

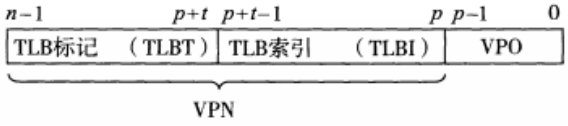


图 9-15 虚拟地址中用以访问 TLB 的组成部分

9.6.3 多级页表

当一级页表中一个PTE是空的，二级页表根本就不会存在，代表着一种巨大的潜在节约。只有一级页表在需要总是在主存中。

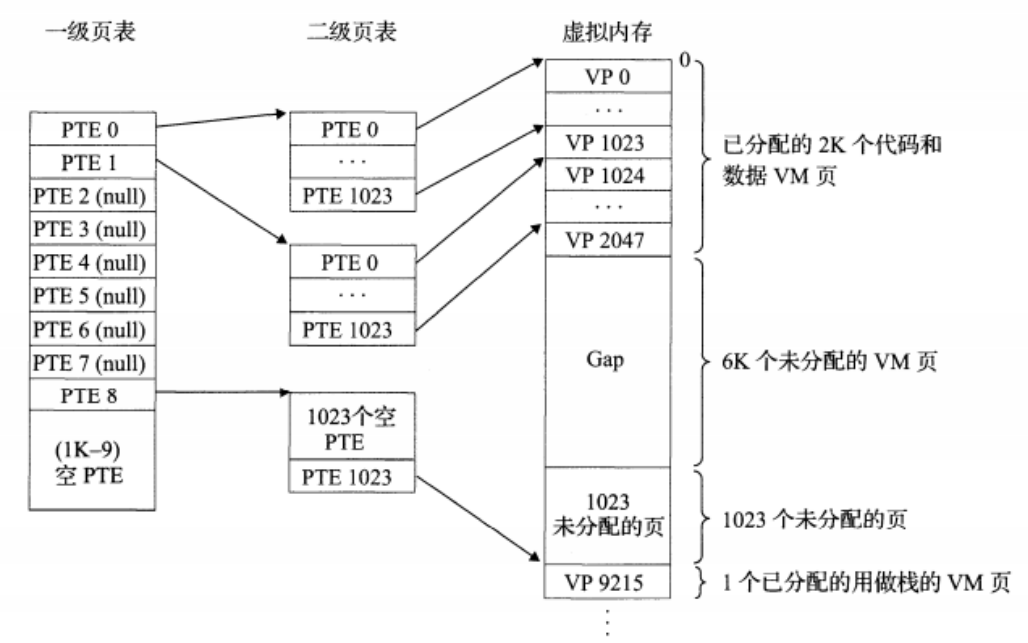


图 9-17　一个两级页表层次结构。注意地址是从上往下增加的

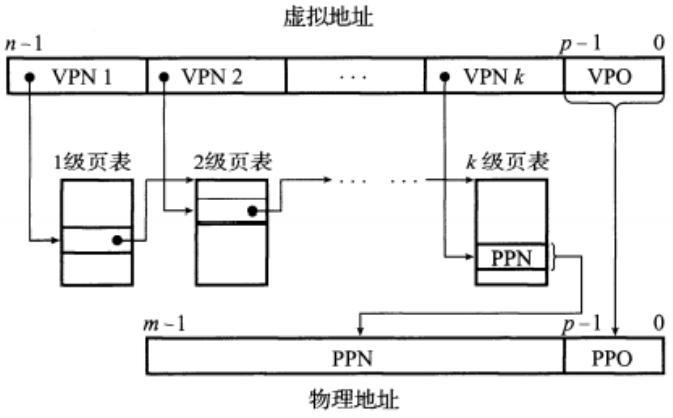


图 9-18　使用 k 级页表的地址翻译

9.6.4 综合：端到端的地址翻译

9.8 内存映射

linux通过将一个虚拟内存区域与一个磁盘上的对象关联起来，以初始化这个虚拟内存区域的内容，这个过程成为内存映射。虚拟内存区域可以映射到两种类型的对象中的一种：

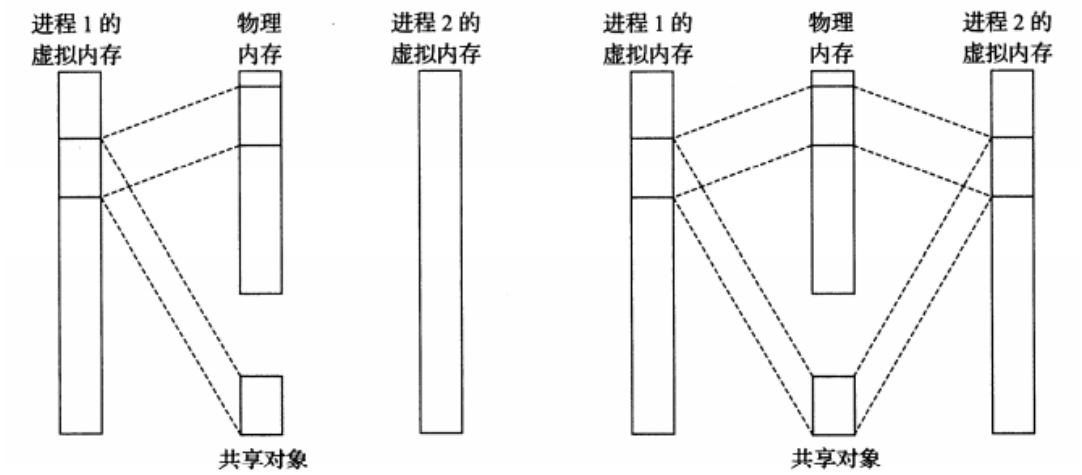
1）linux文件系统中的普通文件：一个区域可以映射到一个普通磁盘文件的连续部分，例如一个可执行目标文件。文件区被分成页大小的片，每一片包含一个虚拟页面的初始内容。

2）匿名文件：匿名文件是由内核创建的，包含的全是二进制零。

无论哪种情况，一旦一个虚拟页面被初始化了，它就在一个由内核维护的专门的交换文件之间换来换去。交换文件也叫做交换空间或者交换区域。交换区间限制着当前运行进程能够分配的虚拟页面的总数。

9.8.2 共享对象

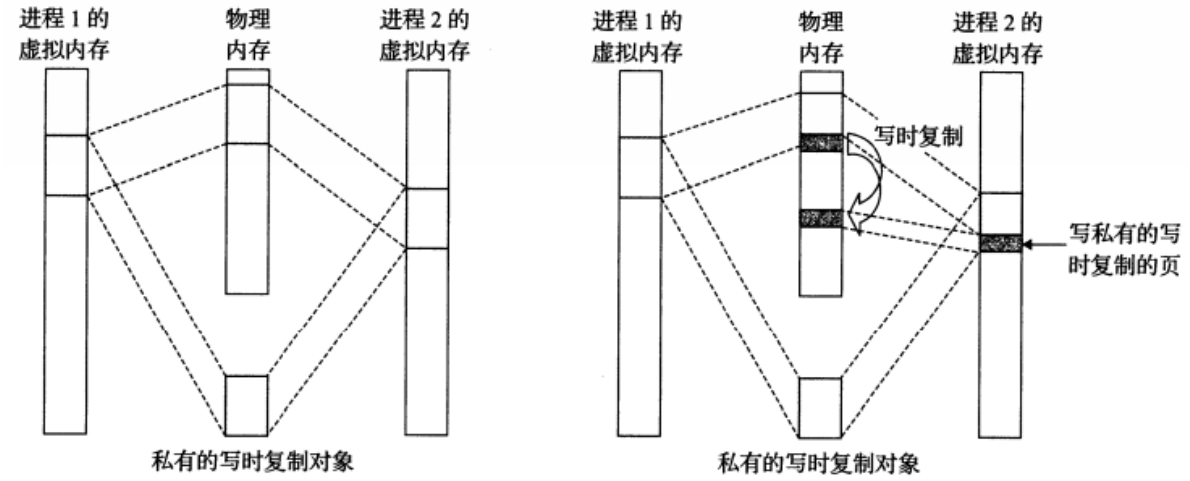
内存中只保存共享对象的一个副本



a）进程 1 映射了共享对象之后　　b）进程 2 映射了同一个共享对象之后

图 9-29　一个共享对象（注意，物理页面不一定是连续的）

私有对象写时复制



a）两个进程都映射了私有的写时复制对象之后　　b）进程 2 写了私有区域中的一个页之后

图 9-30　一个私有的写时复制对象

9.8.2 fork函数

给子进程创建虚拟内存，为当前进程的mm_struct、区域结构和页表的原样副本。
将每个页面都标记为只读，并将两个进程中每个区域结构都标记为私有的写时复制。

9.8.3 execve函数

9.8.3 使用mmap函数的用户级内存映射

Linux 进程可以使用 mmap 函数来创建新的虚拟内存区域，并将对象映射到这些区域中。

```
#include <unistd.h>
#include <sys/mman.h>

void  *mmap(void *start, size_t length, int prot, int flags,
            int fd, off_t offset);
                        返回：若成功时则为指向映射区域的指针，若出错则为 MAP_FAILED(-1)。
```

mmap 函数要求内核创建一个新的虚拟内存区域，最好是从地址 start 开始的一个区域，并将文件描述符 fd 指定的对象的一个连续的片(chunk)映射到这个新的区域。连续的对象片大小为 length 字节，从距文件开始处偏移量为 offset 字节的地方开始。start 地址仅仅是一个暗示，通常被定义为 NULL。为了我们的目的，我们总是假设起始地址为 NULL。图 9-32 描述了这些参数的意义。

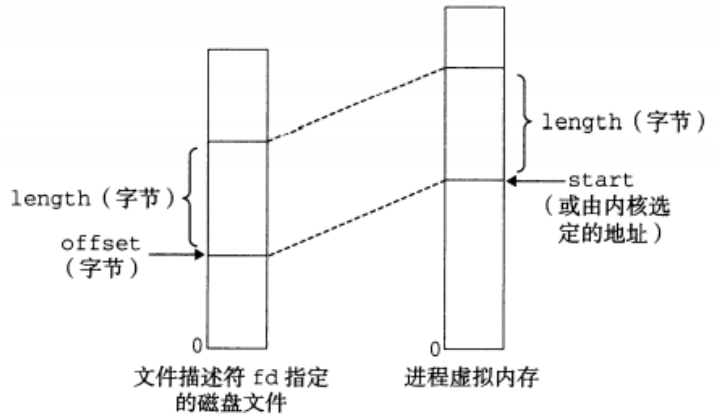


图 9-32　mmap 参数的可视化解释

munmap 函数删除虚拟内存的区域：

```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
```

返回：若成功则为 0，若出错则为 −1。

munmap 函数删除从虚拟地址 start 开始的，由接下来 length 字节组成的区域。接下来对已删除区域的引用会导致段错误。

9.9 动态内存分配

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。

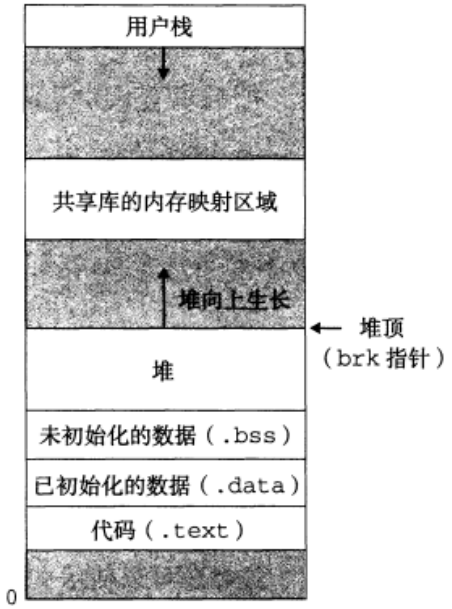


图 9-33　堆

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。

分配器有两种基本风格。两种风格都要求应用显式地分配快。他的不同之处在于由哪个实体来负责释放已分配的块。

- 显示分配器：要求应用显式地释放任何已分配地块。
- 隐式分配器：当分配器检测一个已分配块何时不再被程序所使用，就是放这个块。隐式分配器也叫做垃圾收集器，而自动释放未使用的已分配的块的过程叫做垃圾收集。

9.9.1 malloc和free函数

malloc返回的内存块可能包含这个块内的任何数据对象类型做对齐。32位位模式中，malloc返回的块的地址总是8的倍数。在64位模式中，该地址总是16位地倍数。calloc将分配的内存初始化为零。想要改变一个以前已分配块的大小，可以使用realloc函数。

```
#include <unistd.h>

void *sbrk(intptr_t incr);
                                                    返回：若成功则为旧的 brk 指针，若出错则为－1。
```

sbrk 函数通过将内核的 brk 指针增加 incr 来扩展和收缩堆。如果成功，它就返回 brk 的旧值，否则，它就返回－1，并将 errno 设置为 ENOMEM。如果 incr 为零，那么 sbrk 就返回 brk 的当前值。用一个为负的 incr 来调用 sbrk 是合法的，而且很巧妙，因为返回值(brk 的旧值)指向距新堆顶向上 abs(incr)字节处。

负的incr可以用来释放堆。

```
#include <stdlib.h>

void free(void *ptr);
                                                    返回：无。
```

free函数没有返回值，所以就不会告诉应用程序出现了错误，就会出现一些令人迷惑的运行时错误。

9.9.2 为什么要使用动态内存分配

运行时才能知道某些数据结构的大小

9.9.3 分配器的要求和目标

显示分配器必须在一些相当严格地约束条件下工作

- 处理任意请求序列：一个应用可以有任意地分配请求和释放请求序列，只要满足约束条件：每个释放请求必须对应于一个当前已分配块，这个块是由以前地的分配请求获得的。
- 立即响应请求：分配器必须立即响应分配请求，不允许使用重新排列或者缓冲请求
- 只使用堆：分配器必须对齐块，使得他们可以保存任何类型的数据对象。
- 不修改已分配的块
  在这些限制条件下，吞吐量最大化和内存使用率最大化，性能目标是相互冲突的。

9.9.4 碎片

内部碎片和外部碎片

9.9.5 实现问题

9.9.6 隐式空闲链表

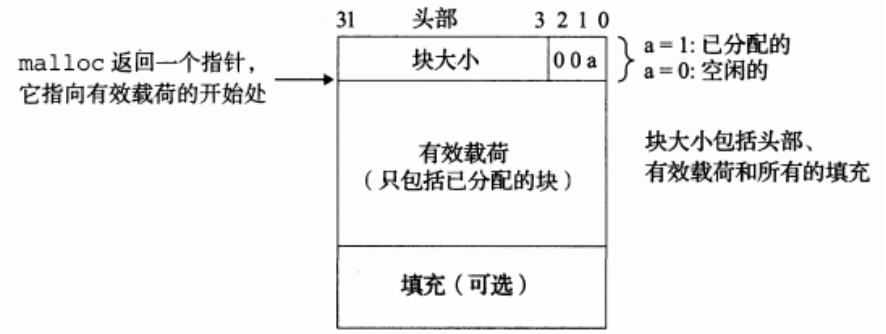


图 9-35　一个简单的堆块的格式

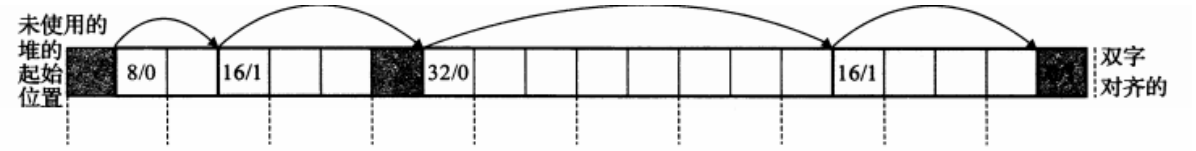


图 9-36　用隐式空闲链表来组织堆。阴影部分是已分配块。没有阴影的部分是空闲块。
头部标记为(大小(字节)/已分配位)

因为空闲块是通过头部中的大小字段隐含地连接着的。
优点是简单，缺点是任何操作的开销。另一个缺点是最小块大小（因为要有头部且有对齐要求）

9.9.7 放置已分配的块

- 首次适配：从头开始搜索，选择第一个合适的空闲块。优点是它趋于将大的空闲块保留在链表的后面
- 下一次适配：从上一次查询结束的地方。速度快，但是空间利用率最低。
- 最佳适配：内存块大小从小到大排列，选择第一个大于的。空间利用率最高。

9.9.8 分割空闲块

9.9.9 获取额外的堆空间

如果分配器不能为请求找到合适的空闲块，一个选择是合并空闲块，另一个是分配器就会通过调用sbrk函数，向内核请求额外的堆内存。分配器将额外的内存转化成一个大的空闲块，将这个块插入到空闲链表中，然后将被请求的块放置在这个新的空闲块中。

9.9.10 合并空闲块

分配器可以选择立即合并，也可以选择推迟合并（直到某个分配请求失败，然后扫描整个堆，合并所有的空闲块）
立即合并可能会造成一种形式的抖动。

9.9.11 带边界标记的合并

因为合并前面的块有困难，所以使用边界标记，在每个块的结尾处添加一个脚部，脚部是头部的一个副本。

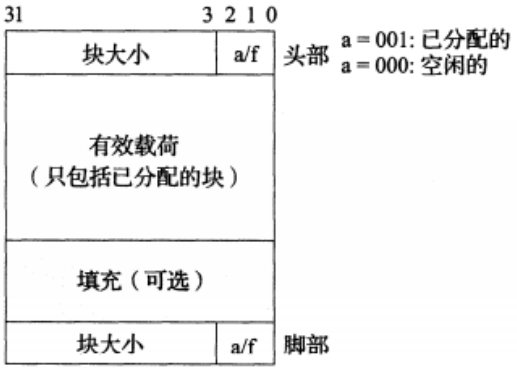


图 9-39　使用边界标记的堆块的格式

9.9.12 综合：实现一个简单的分配器

p597开始

9.9.13 显示空闲链表

空闲块包含指向前驱空闲块和后继空闲块的指针。

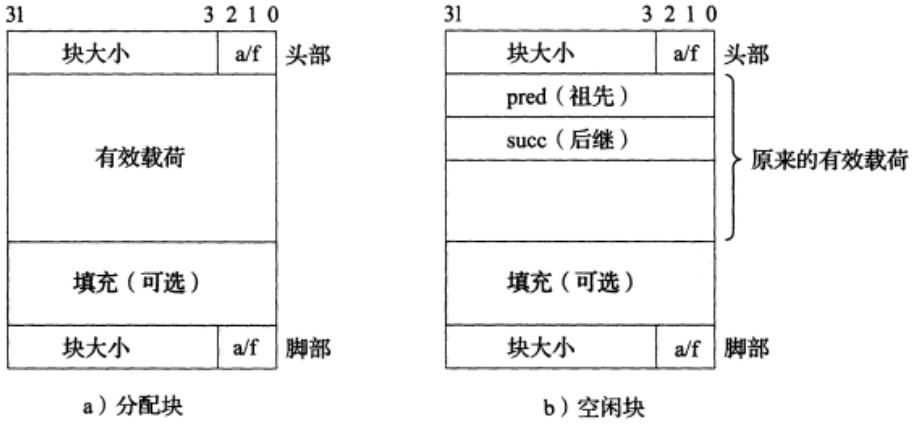


图 9-48　使用双向空闲链表的堆块的格式

可以使首次分配时间从块总数的线性时间降低至空闲块数量的线性时间（因为只用显式链表查询空闲块）。
释放块的时间可以是在常数时间内完成也可以是线性的。
一种方法是使用后进先出的顺序维护链表，将新释放的块放置在链表的开始位置，使用LIFO的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块（因为后进先出，最后进的都是刚释放的块），这种情况下释放一个块可以在常数时间内完成（因为只要删除直接加到链表尾，这一次操作只要一个常数时间）。如果使用了边界标记，那么合并也可以在常数时间内完成。
另一种方法是按照地址顺序来维护链表，其中链表中每个块地址都小于它后继的地址。这种情况下，释放一个块需要线性时间来搜索定位合适的前驱（用以插入刚释放的块）。优点是内存利用率更高。

9.9.14 分离的空闲链表

分离存储，维护多个空闲链表，每个链表中的块有大致相等的大小。一般思路是将所有可能的块大小分成一些等价类，也叫做大小类。每个大小类一个空闲链表，按照大小的升序排列。当分配器需要一个大小为n的块时，他就搜索相应的空闲链表。如果不能找到适合的块与之匹配就搜索下一个链表。

- 简单分离存储
  每个大小类的空闲链表包含大小相等的块。为了分配一个给定大小的块，我们检查相应的空闲链表，如果链表非

空，我们简单地分配其中第一个块的全部。空闲块是不会分割以满足分配请求的。如果链表为空，分配器就向OS请求一个固定大小的额外内存片，将这个片分成大小相等的块，并将这些块链接起来形成新的空闲链表。释放一个只需将这个块插到相应的空闲链表的前部。

- 分离适配
  找到一个适合的块后，分割该块，将剩余的部分插入到适当的空闲链表中。若要释放一个块，执行合并，并将合并后的块放置到相应的空闲链表中。
- 伙伴系统
  每一次都二分存储空间

3. 伙伴系统

伙伴系统(buddy system)是分离适配的一种特例，其中每个大小类都是 2 的幂。基本的思路是假设一个堆的大小为 2^m 个字，我们为每个块大小 2^k 维护一个分离空闲链表，其中 $0 \leqslant k \leqslant m$。请求块大小向上舍入到最接近的 2 的幂。最开始时，只有一个大小为 2^m 个字的空闲块。

为了分配一个大小为 2^k 的块，我们找到第一个可用的、大小为 2^j 的块，其中 $k \leqslant j \leqslant m$。如果 $j=k$，那么我们就完成了。否则，我们递归地二分割这个块，直到 $j=k$。当我们进行这样的分割时，每个剩下的半块(也叫做伙伴)被放置在相应的空闲链表中。要释放一个大小为 2^k 的块，我们继续合并空闲的伙伴。当遇到一个已分配的伙伴时，我们就停止合并。

关于伙伴系统的一个关键事实是，给定地址和块的大小，很容易计算出它的伙伴的地址。例如，一个块，大小为 32 字节，地址为：

$$xxx\cdots x00000$$

它的伙伴的地址为

$$xxx\cdots x10000$$

换句话说，一个块的地址和它的伙伴的地址只有一位不相同。

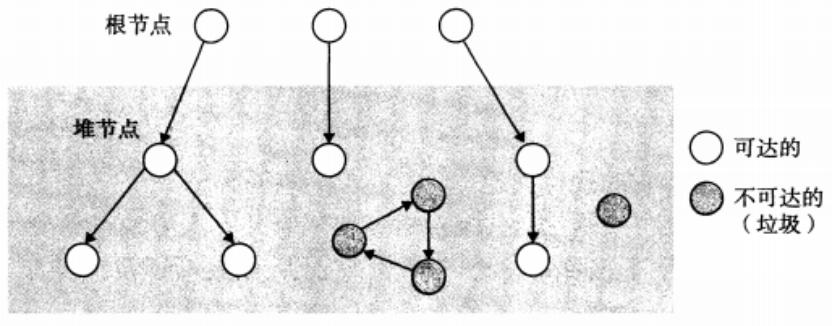
伙伴系统分配器的主要优点是它的快速搜索和快速合并。主要缺点是要求块大小为 2 的幂可能导致显著的内部碎片。因此，伙伴系统分配器不适合通用目的的工作负载。然而，对于某些特定应用的工作负载，其中块大小预先知道是 2 的幂，伙伴系统分配器就很有吸引力了。

9.10 垃圾收集

垃圾收集器是一种动态内存分配器，他自动释放程序不再需要的已分配块。

9.10.1 垃圾收集器的基本知识

垃圾收集器将内存视为一张有向可达图。



有向边P->Q意味着块P中的某个位置指向块Q中的某个位置。
诸如C++的收集器通常不能维持可达图的精确表示。这样的垃圾收集器叫做保守的垃圾收集器。即每个可达块都被正确的标记了，而一些不可达节点却被错误的标记为可达。

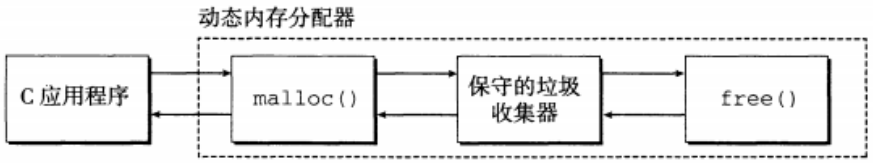


图 9-50　将一个保守的垃圾收集器加入到 C 的 malloc 包中

无论何时需要堆空间时，应用都会用通常的方式调用 malloc。如果 malloc 找不到一个合适的空闲块，那么它就调用垃圾收集器，希望能够回收一些垃圾到空闲链表。收集器识别出垃圾块，并通过调用 free 函数将它们返回给堆。关键的思想是收集器代替应用去调用 free。当对收集器的调用返回时，malloc 重试，试图发现一个合适的空闲块。如果还是失败了，那么它就会向操作系统要求额外的内存。最后，malloc 返回一个指向请求块的指针(如果成功)或者返回一个空指针(如果不成功)。

9.11 C程序中常见的与内存有关的错误

- 间接引用坏指针
- 读未初始化的内存
- 允许栈缓冲溢出
- 假设指针和他们指向的对象是相同大小的
- 造成错位错误
- 引用指针，而不是它所指向的对象（*的优先级和+-一样高）
- 指针运算是以它们指向的对象的大小为单位来进行的
- 引用不存在的变量
- 引用空闲堆块中的数据。
- 引起内存泄漏