《深入理解计算机系统》| 虚拟存储器

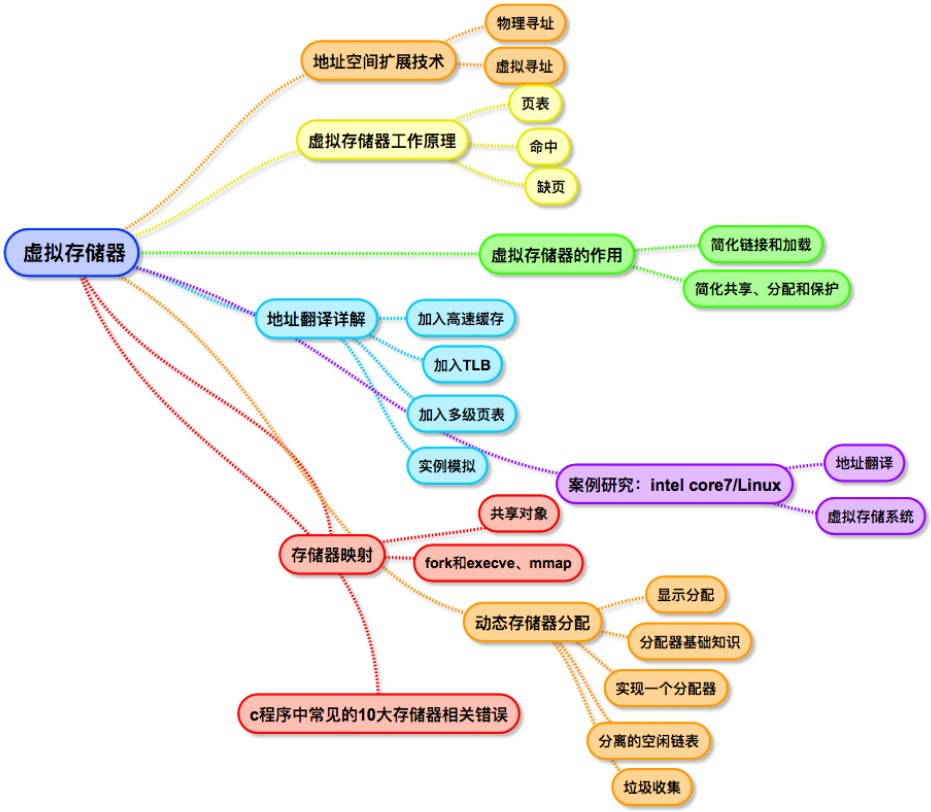
唐鱼的学习探索 关注

0.55 2018.04.19 13:07:35 字数 10,679 阅读 2,832

唐鱼的学习探索 关注
总资产29 (约2.85元)

如何高效的准备一次考试
阅读 1,737

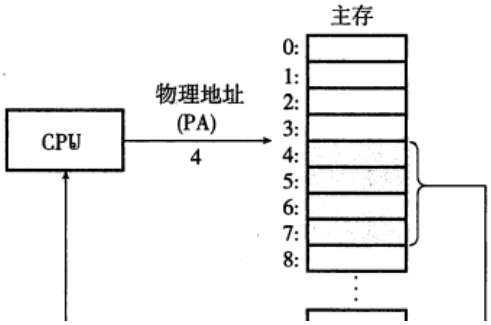
都9102年了，你还不知道anki是什么
阅读 102



虚拟存储器又叫做虚拟内存，我们现在的操作系统普遍都支持了虚拟内存，这样做是因为我们同时运行着太多的程序了，就目前我电脑的状态来看，我既要打开浏览器，又要听歌，可能同时还登陆的有QQ，如果不使用虚拟内存4G的内存空间很快就会被耗尽，而一旦没有了内存空间，其他程序就无法加载了。虚拟内存的出现就是为了解决这个问题，当一个程序开始运行的时候，其实是为每个程序单独创建了一个页表（这个以后讲），只将一部分放入内存中，以后根据实际的需要随时从硬盘中调入内容。当然虚拟内存不仅仅只有这个功能，我们的操作系统也是在内存中运行着的，虚拟内存同时还提供了一种保护，这样做其他进程就不会损坏掉系统的内存空间。那么虚拟内存是如何实现的呢？

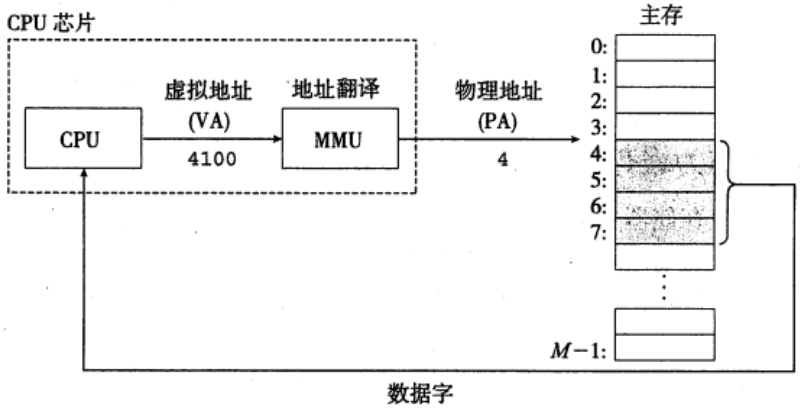
1.1 物理寻址和虚拟寻址

虚拟内存主要是一种地址扩展技术，主要是建立和管理两套地址系统：物理地址和虚拟地址。由虚拟地址空间（硬盘上）装入进程，其实际执行是在物理地址空间（内存上）承载进程的执行。虚拟地址空间比物理地址空间要大的多，操作系统同时承担着管理者两套地址空间的转换。我们来看看什么是物理寻址：



主存的每个地址都是唯一的，第一个字节地址为0，接下来为2，以此类推。CPU使用这种访问方式就是**物理寻址**。上图所示就是CPU通过**地址总线**传递读取主存中4号地址开始处的内容并通过数据总线传送到CPU的寄存器中。

当然地址总线也不是无限大的，我们通常所说的32位系统，其寻址能力是2^32 = 4 294 967 296B（4GB）也就是说内存条插的再多也没有用，地址总线只能最多访问到4GB的地址内容。我们前面说过4GB的物理内存空间其实并不大（如果是独占的话）。这时候科学家们想到了一个很好的方法，建立虚拟寻址方式，使用一个成为MMU的地址翻译工具将虚拟地址翻译成物理地址在提供访问，如下图：



使用虚拟寻址的时候，cpu先是生成一个虚拟地址：4100再经过地址翻译器，将4100翻译成物理地址。

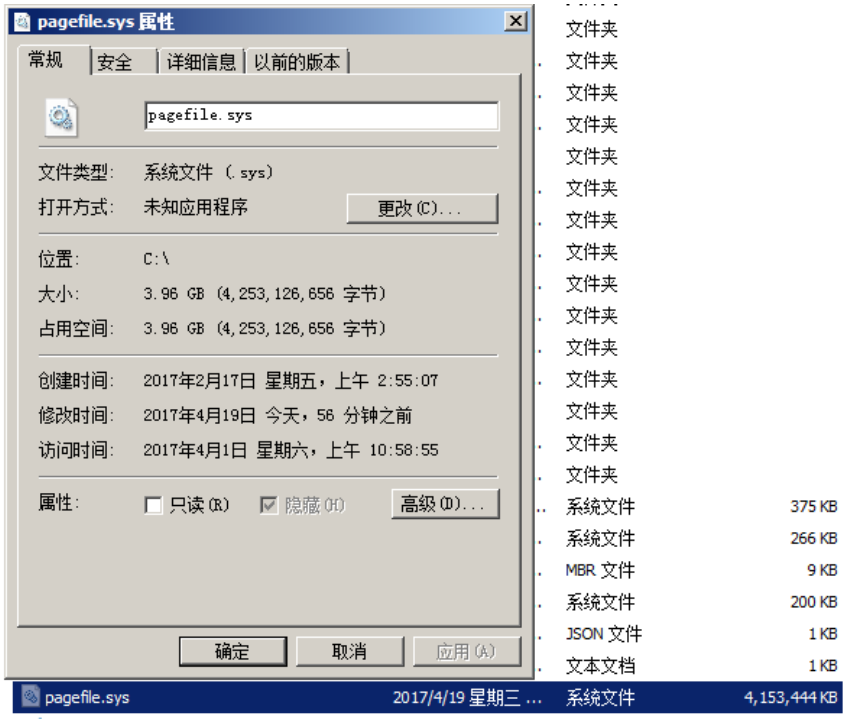
我们说过虚拟地址要比物理地址大的多，为啥还要麻烦的将物理地址转成虚拟地址呢？虚拟地址的发明究竟是为了什么，我们知道对内存的访问要比硬盘的访问快10000倍，如果我们在内存中没有找到相应的内容（不命中），而需要到硬盘上找的话，我们必须要提供相对来说高效率的访问方式。这时候就创建了一个虚拟存储器，管理着磁盘，以每页的方式进行整合，每个页面的大小4kb-2mb不等，加上偏移量就成为了一个虚拟地址。比如4100，说明的就是页4编号，偏移100处的位置。这就比挨个挨个单独寻址要快的多。

1.2 地址空间

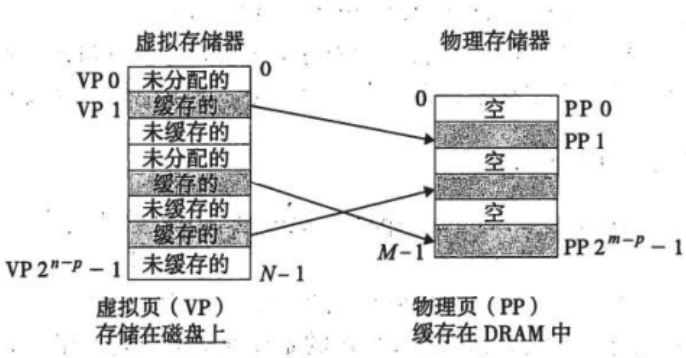
地址空间是一个非负整数的集合{0，1，2，.....}，一个32位的系统中有：2^32 = 4 294 967 296B（4GB）个有效地址。地址空间的概念很重要，我们必须要清楚数据对象（字节）和它的属性（地址）的区别， 举个例子：我和我老婆住在苍溪县xx小区7栋1单元，这个就是我的属性：地址。另外，住在家的我和我老婆就是数据对象（字节）。虚拟存储器的基本思想是：主存中的每个字节都有一个选自虚拟地址空间的虚拟地址和一个选自物理地址空间的物理地址。

1.3 虚拟存储器的工作原理

我们先来看看虚拟内存，就windows系统而言是保存在磁盘上的一个文件，存放于C盘的pagefile.sys点击属性可以看到其大小为3.96G，这相当于一个仓库，保存着临时需要又还没用到的数据。



这个仓库的的数据被分割成块，称为虚拟页。虚拟存储器的主要思想就是：在主存中缓存硬盘上的虚拟页（pagefile.sys），虚拟页有三个状态：未分配、缓存的、未缓存的。



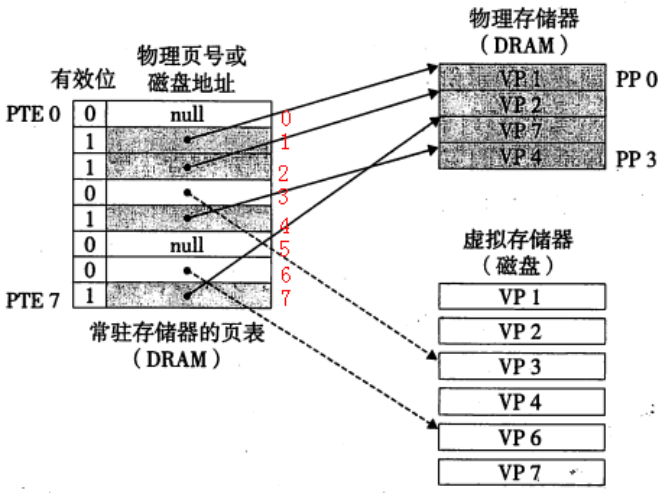
上图所示的是一个有8个虚拟页的小虚拟存储器（建立在硬盘上），虚拟页0和3还未分配，因此在磁盘上还不存在。虚拟页1、4和6被缓存在右边的主存中。

(内存访问速度要比硬盘快10000倍，因此不命中的话代价要昂贵的多。我们前面说过是以虚拟页来缓存的，也就是分成块，每个块（虚拟页）的大小4kb-2mb不等。)

我们现在来看看地址翻译MMU是如何完成虚拟地址到物理地址的转换的，学习这个知识是帮助我们理解虚拟存储器是如何将虚拟也缓存到主存（内存）中去的。

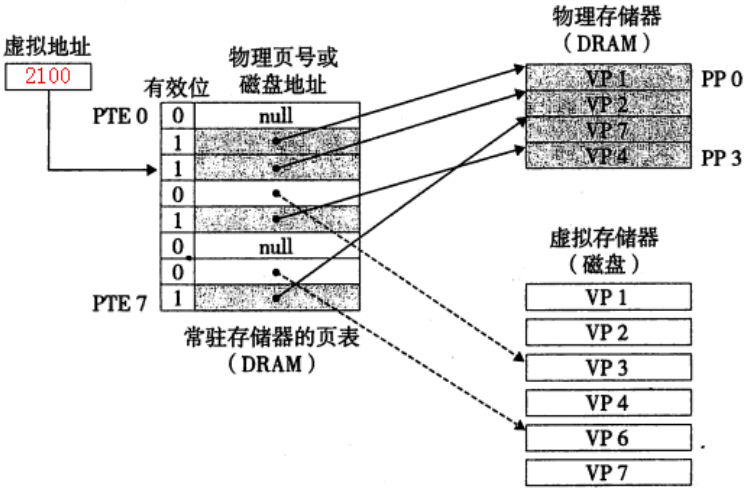
① 页表

页表是一个存放在内存中的数据结构，MMU就是通过页表来完成虚拟地址到物理地址的转换。这个数据结构每一个条目称为PTE（Page Table Entry），由两部分组成：有效位和n位地址段。有效位如果是1，那么n位地址就指向已经在内存中缓存好了的地址；如果为0，地址为null的话表示为分配，地址指向磁盘上的虚拟内存（pagefile.sys）的话就是未缓存。我们来看一个典型的页表图：



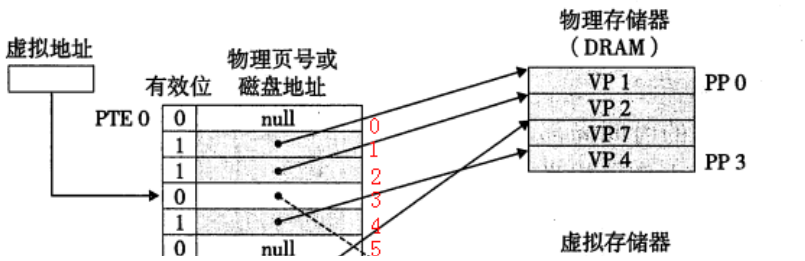
虚拟页vp1,2,7,4当前被缓存在内存中，页表上有效位设置成1，分别用PTE1，2，4，7表示。VP0和VP5（PTE0、5）未被分配，VP3和VP6被分配并指向虚拟内存，但未被缓存。

② 页命中

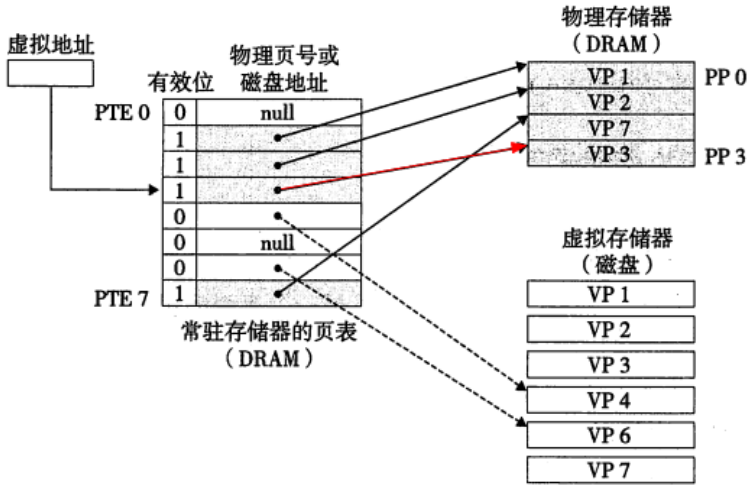


当我们使用2100虚拟地址来访问虚拟页2的内容的时候，就是一个页命中。地址翻译将指向PTE2上，由于有效位1，地址翻译器MMU就知道VP2已经缓存在内存中了。就使用页表中保存的物理地址进行访问。

③ 缺页



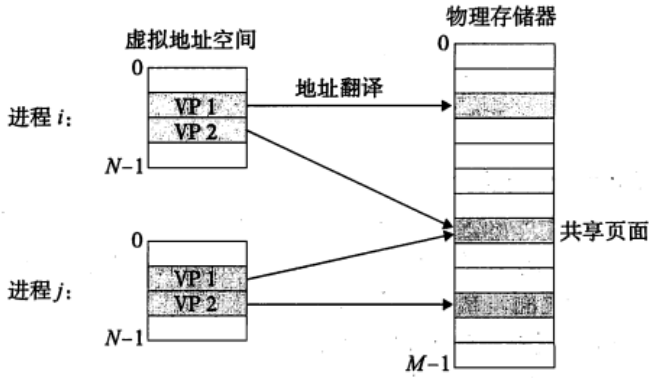
我们再来看看不命中，也就是缺页的情况，当CPU需要VP3的一个字时，初始化是这样的：



PTE3有效位是0，同时地址位指向了虚拟内存（pagefile.sys），就会触发缺页异常。异常处理程序会选择牺牲一个内存（DRAM）中的页，本例中选择的是内存中的PP3页的VP4，接下来内核就从虚拟内存中拷贝VP3到内存中的PP3，并使得PTE3指向内存中的PP3，形成如下：

（注：虚拟存储器出现早于高速缓存，按照习惯的说法块被叫做页。从虚拟内存到物理内存传送页的活动就叫做页面交换。）

1.4 虚拟存储器的作用



虚拟存储器有诸多的好处，操作系统其实为每个进程提供了一个独立的页表，使用不同的页表也就创建了独立的虚拟地址空间，下图展示了基本思想：

进程i将VP1映射到了内存的PP2处，VP2映射到了内存的PP7处。进程j将VP1映射到了内存的PP7，将VP2映射到了PP10处。

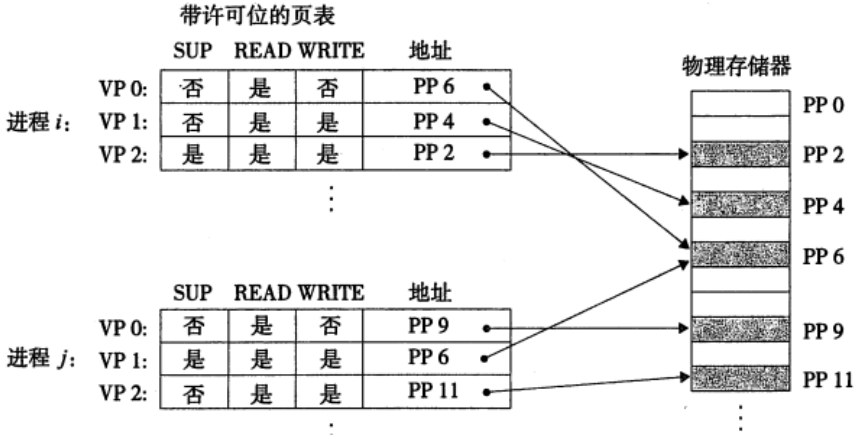
简化链接：每个进程一个页表后，这个进程就会觉得全世界都是它的（页表模拟出一个虚拟存储器），那什么符号链接的时候（也就是符号映射到地址的时候），不再会受到内存中还有其他应用程序的干扰，因为我们面向的是虚拟存储器，我们的进程的地址空间是独立的，我这个符号放到离0偏移100的地方，那个放到离0偏移200的地方很容易就搞定了。

简化加载：在硬盘中双击一个图标，启动一个应用程序时，实际上你都不需要将这个程序从硬盘给加载到内存，只需要建个页表，然后页表里的编号指向的是硬盘，然后CPU访问到具体代码的时候，再按照上一节的寻址的方式，按需的将硬盘上的东东加载到内存。加载过程及其简单了。

简化共享：我们有很多的进程在系统中运行，但是有些代码，比如调用操作系统的API，这些API可能许多进程都要使用比如printf，这就要共享一部分内存，我们不需要将这部分内存在每个进程空间都拷贝一份，实际上每个进程都有一个页表，而不是全局只有一个，页表把共享内存映射到同一个地方。

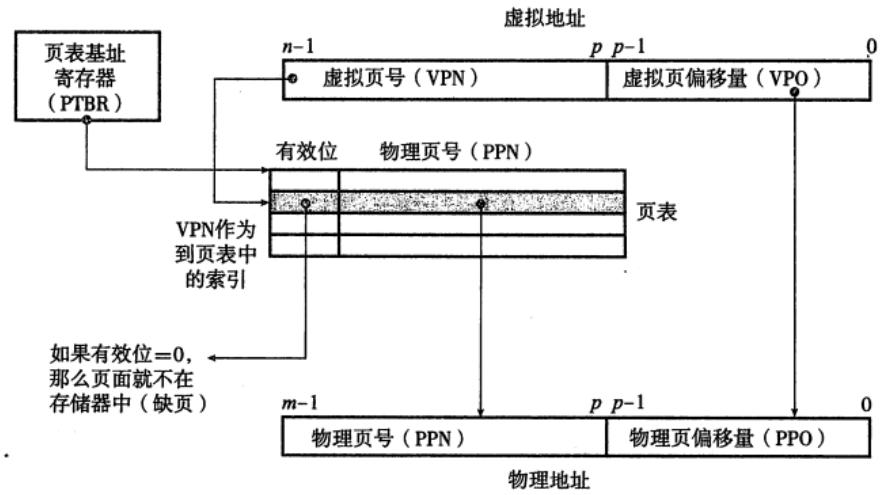
简化存储器分配：当一个进程使用malloc要求额外的空间时，操作系统只需要保证形成了一个连续的虚拟页面，但可以映射到物理内存中任意的位置，可以随机分散在内存的不同位置。

简化保护：我们可以通过为PTE添加额外的标识位提供对存储器的保护。



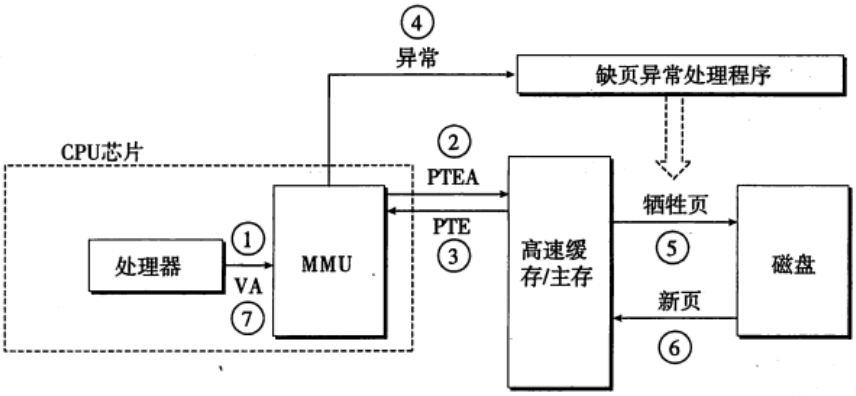
通过新添加的三个标识位：SUP：内核or用户；READ:读；WRITE：写。运行在用户模式下的进程只允许访问SUP为否的页面，如果一个指令违法了访问的设置条件，就会转到保护故障，引起一个段错误。

1.5 虚拟存储器工作原理详解：地址翻译



地址翻译从形式上来说就是建立一个虚拟地址空间到物理地址空间的映射关系，我们前面说过MMU使用的是页表来实现这种映射。CPU中有一个专门的页表基址寄存器（PTBR）指向当前页表，使用页表进行翻译的时候方法如下：

每个虚拟地址由两部分组成：虚拟页号（VPN）+虚拟页偏移量（VPO），当CPU生成一个虚拟地址并传递给MMU开始翻译的时候，MMU利用虚拟地址的VPN来选择相应的PTE，同时将页表中的物理页号（PPN）+虚拟地址的VPO就生成了相应的物理地址。（**物理地址是由页表中的物理页号+虚拟地址中的偏移量构成**）

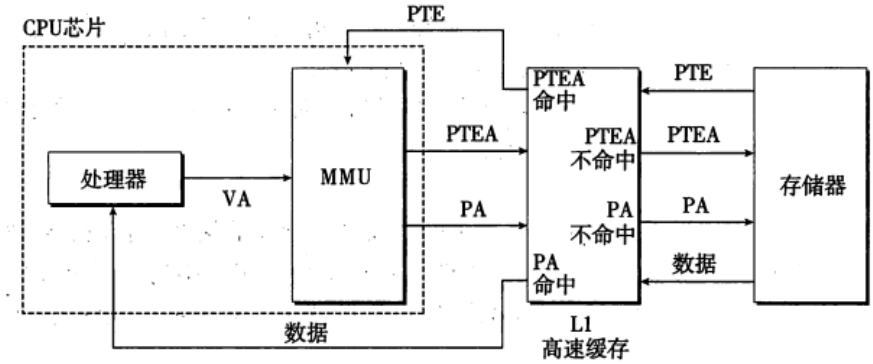


页面命中是一个简单的过程，我们就不做详解，这里来跟踪看一下**缺页的情况**：

> 说明：①CPU生成虚拟地址；②MMU生成PTE地址从内存的页表中请求内容；③ 内存中的页表返回相应的PTE值；④ PTE的有效位是0，转到异常处理程序；⑤ 异常处理程序确定内存中的牺牲页，并将其写会到磁盘上（pagefile.sys）；⑥从pagefile.sys中调入新的文件并更新PTE。⑦ 由于PTE已经被更新好了，从新发送虚拟地址到MMU（后面就和命中的过程一样了）

我们讲了大致的地址翻译原理，有什么办法能够提高翻译的速度吗？

① 加入高速缓存

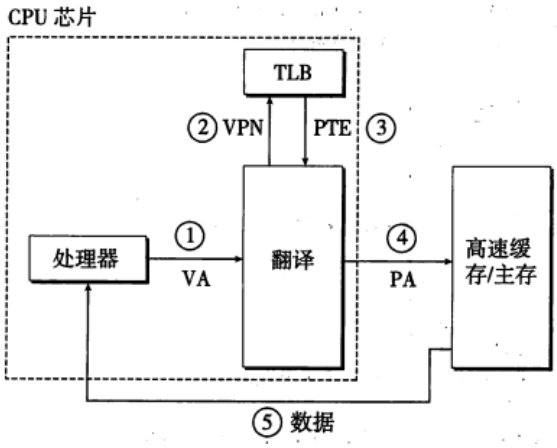


高速缓存被发明出来的一个重要原因就是提高对内存的访问速度，我们来看看加入高速缓存后的访问示意图：

高速缓存被放在存储器和MMU之间，可以缓存页表条路。当MMU发送一个PTEA请求的时候，优先从高速缓存中寻找相应的PTE值，如果命中直接返回给MMU，如果不命中从内存中获得并发送到高速缓存，再由高速缓存返回到MMU。（高速缓存使用的是物理寻址，不涉及地址保护

问题，因为MMU已经加入了保护标识位）

② 加入翻译后备缓冲器TLB

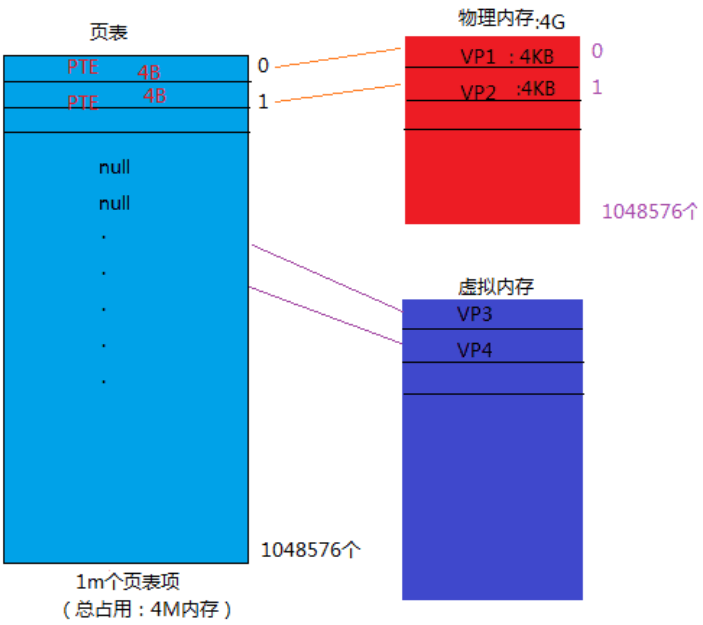


TLB是一个小的、虚拟寻址的缓存，其中每一行都保存一个PTE块，高度相连。主要是提供虚拟地址到物理地址的翻译速度。大致范围示意图如下：

> 说明：①CPU生成一个虚拟地址并发送到MMU；② ③MMU从TLB中获取相应的PTE④翻译成相应的物理地址后从内存中请求内容；⑤ 数据从内存返回给CPU

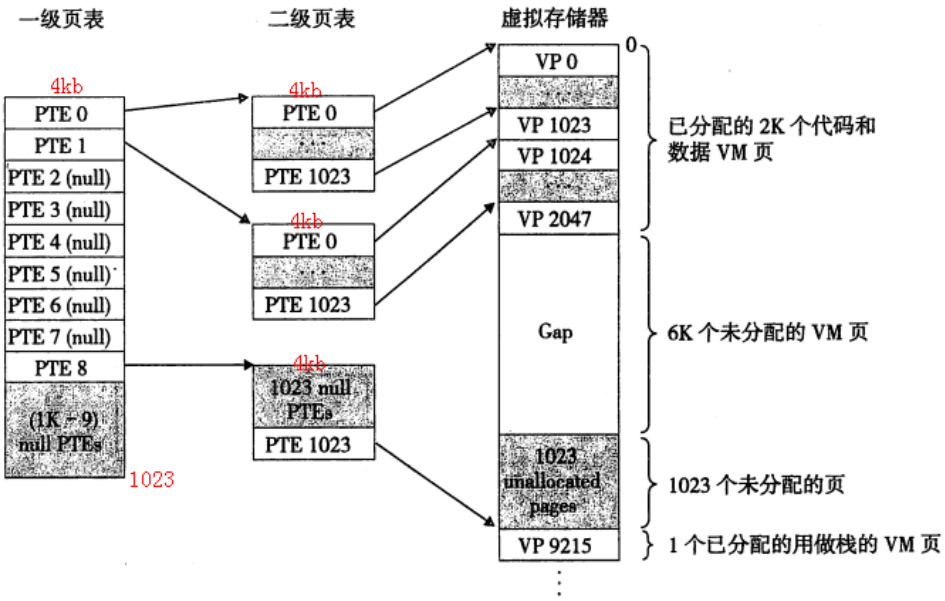
③ 加入多级页表

我们来分析一下单级页表的弱势之处，然后指出改进的方法。我们双击图标运行一个程序的时候，在单级页表模式下，其实是在内存中为这个程序创建了一个页表，使得程序有了独立的地址空间。我们以32位系统4GB地址空间为例，我们将物理内存分割为虚拟的页面，每个页面保存4KB大小的内容，这样我们总共需要1048576个页面，才能瓜分所有的4GB空间。那么我们的页表要能够完成所有物理内存的映射，就必须要1048576个页表项，由于每个页表项占用4B的空间，那么我们这个页表就需要占用4194304B（4M）的内存空间，每个进程都有这样的一个4M的页表占用着内存空间，才能完成映射。



单级页表

我们来看看有什么方法优化一下，下面我们加入分级页表（二级）：

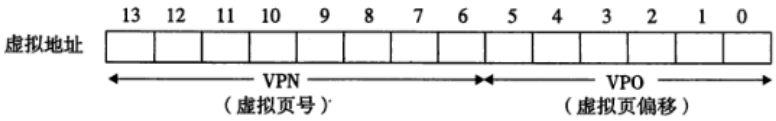


二级页表

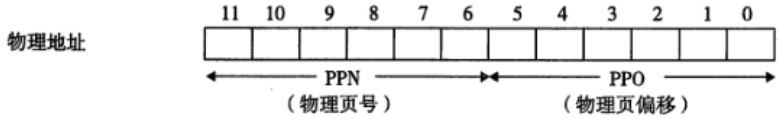
我们加入分级的思想以后，每一级的页表就都只有4KB的大小，数量也有原来的1048576变成了1024个，两级相乘其实表示的数量还是原来那么多。上图所示，一级页表每条PTE负责映射二级页表1024个PTE项，二级页表的每个PTE在映射虚拟存储器中4KB大小的位置。也就是说一级页表每条PTE负责映射一块4M大小的空间，而一级页表总共有1024个页表项，也就能用来映射完成所有物理内存空间。这样做的好处是，如果一级页表中有未被分配的项目，那么这条PTE直接设置成null，不指向任何二级列表，也就不再占用空间。还有一个好处是不是所有的二级列表都需要常驻内存，每个进程只需要在内存中建立一级页表（4kb）大小，二级列表按需要的时候创建调入，这样就更省了。

④ 综合：一个从虚拟地址到物理地址并获取数据的模拟

为了方便讨论，我们以一个小的存储系统作如下假设：



1> 虚拟地址大小14位：结构如下

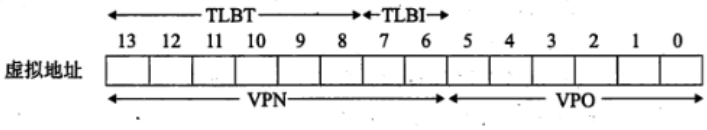


2> 物理地址大小12位：结构如下

VPN	PPN	有效位
00	28	1
01	—	0
02	33	1
03	02	1
04	—	0
05	16	1

VPN	PPN	有效位
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1

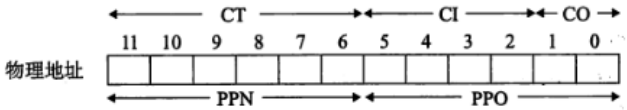
3> 内存大小为4KB，物理页号为64个，每个页面大小为64B，页表如下：



位	标记位	PPN	有效位	标记位	PPN	有效位	标记位	PPN	有效位	标记位	PPN	有效位
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

a）TLB：四组，16个条目，四路组相联

4> TLB 翻译后备缓存器分成4组，每组4条，一共16个条目：



索引	标记位	有效位	块 0	块 1	块 2	块 3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

c）高速缓存：16个组，4字节的块，直接映射

5> 高速缓存64B大小，使用物理寻址、直接映射的方式，每行4B，共计16个组：

	TLBT						TLBI						
	0x03						0x03						
位位置	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN								VPO					
	0x0f								0x14					

好了，有了这些假设以后我们来看一下，当CPU读取0x03d4处内容会发生些什么：

此处是虚拟地址，0x03d4二进制表示就为：（0000 1111 0101 00）14位，由于虚拟地址的

低6位用来表示偏移量（每个页面64B大小：2^6=64），剩下的高8位用来表示虚拟页号，一共有128个虚拟页号（2^8）。

我们从虚拟地址中：

1> 抽取出虚拟页号为：0x0f；

TLBT						TLBI	
0x03						0x03	
13	12	11	10	9	8	7	6
0	0	0	0	1	1	1	1

2> 将虚拟页号与TLB进行对比，为了方便，我们形成TLBT标记位，TLBI组索引；

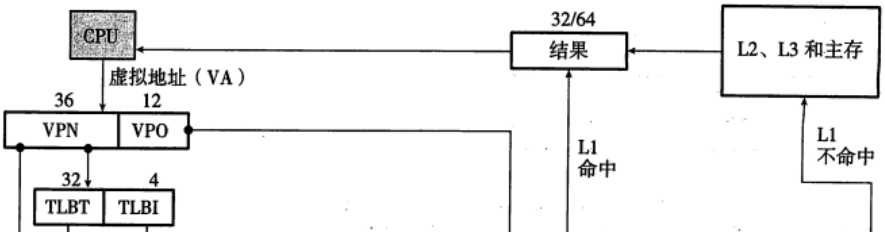
组索引在0x03号位置，标记也为0x03，这时候回到我们的假设“4>”处进行检查，发现0x03组，标记位0x03处的有效位是1，所以命中。取出物理页号（PPN）0D用于构造物理地址用。物理地址就为：PPN-VPO = 0x354：

	CT						CI				CO	
	0x0d						0x05				0x0	
位位置	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

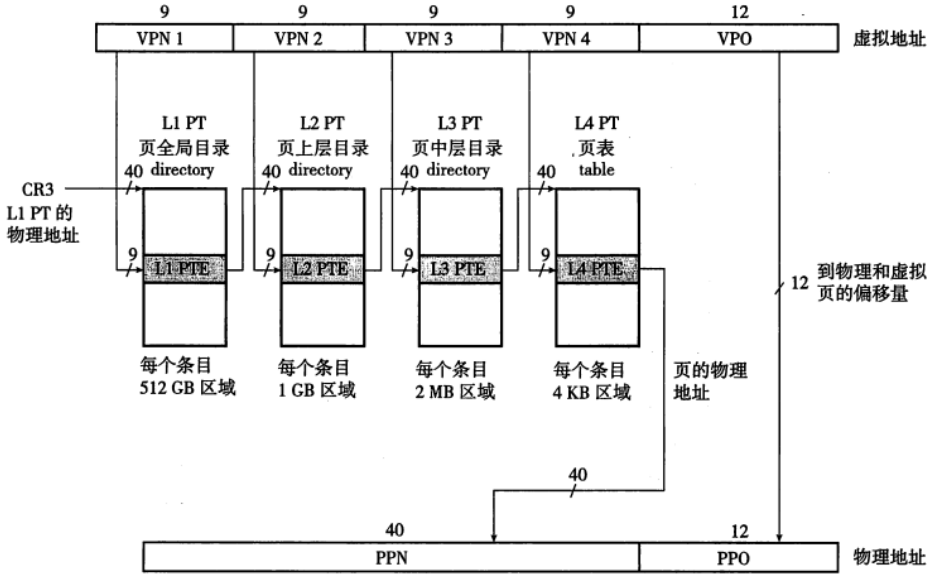
3 > 根据物理地址：0x354，我们在高速缓存中去碰碰运气，前面假设的时候我们说过大小为64B，我们将其分成16个条目，由：标记位+有效位+块0-3组成。其实际存放数据的块每个条目只有4个（0-3）所以总大小为64B，我们的物理地址要到高速缓存中去寻找数据，就得有某种对应方式。其中物理地址的低2位用作偏移量（CO）因为每个条目只有4个数据块，紧接着的4位表示组索引，因为一共是16个组，最后的高7位作为标记位。我们形成如下的：CO=0x0，偏移量为0也就是块0的内容；CI = 0x05也就是第0x05组和CT：0x0d标志位。有了这些内容以后我们返回到假设5中去寻找，发现高速缓存中的5号索引，标记位为0x0d，并且有效，读取块0处的内容为36。这就是我们要返回给CPU的内容。至此完成了一个端到端地址翻译并返回数据的手工模拟，当然我们还可能遇到很多不同的情况。如在高速缓存中不命中，TLB不命中等等，但大致原理几乎类似，请自行脑补。

1.6 案例研究：Intel Corel7/ Linux存储系统

① Core i7 地址翻译：

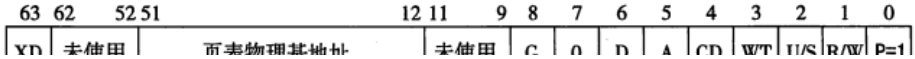


Core i7采用4级页表层次结构，CPU产生的虚拟地址，如果命中由TLB生成物理地址，如果不命中后通过4级页表生成物理地址。物理地址如果命中优先从L1高速缓存中获取数据，如果不命中再从主存中获取结果，最后传递给CPU



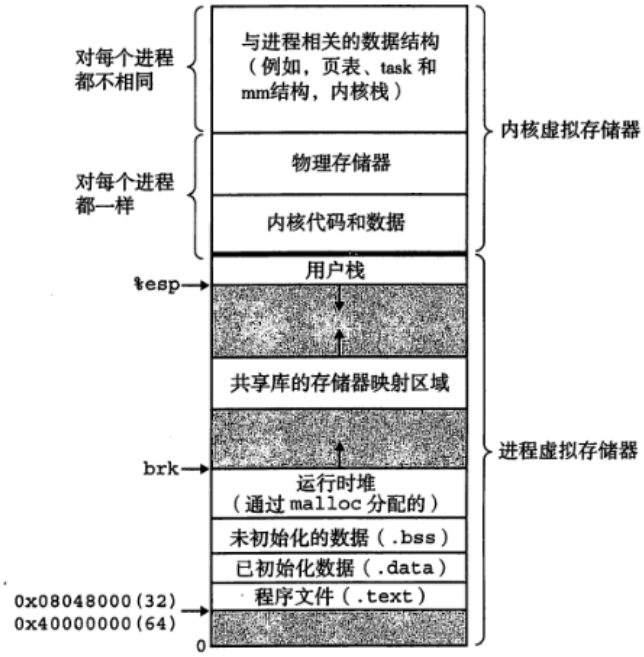
四级页表详解

四级页表将虚拟地址翻译成物理地址的过程也相当简单，36位的虚拟地址被分割成4个9位的片。VPN1有一个到L1 PTE的偏移量，找到这个PTE以后又会包含到L2页表的基础地址；VPN2包含一个到L2PTE的偏移量，找到这个PTE以后又会包含到L3页表的基础地址；VPN3包含一个到L3PTE的偏移量，找到这个PTE以后又会包含到L4页表的基础地址；VPN4包含一个到L4PTE的偏移量，找到这个PTE以后就是相应的PPN（物理页号）。



页表条目格式说明：

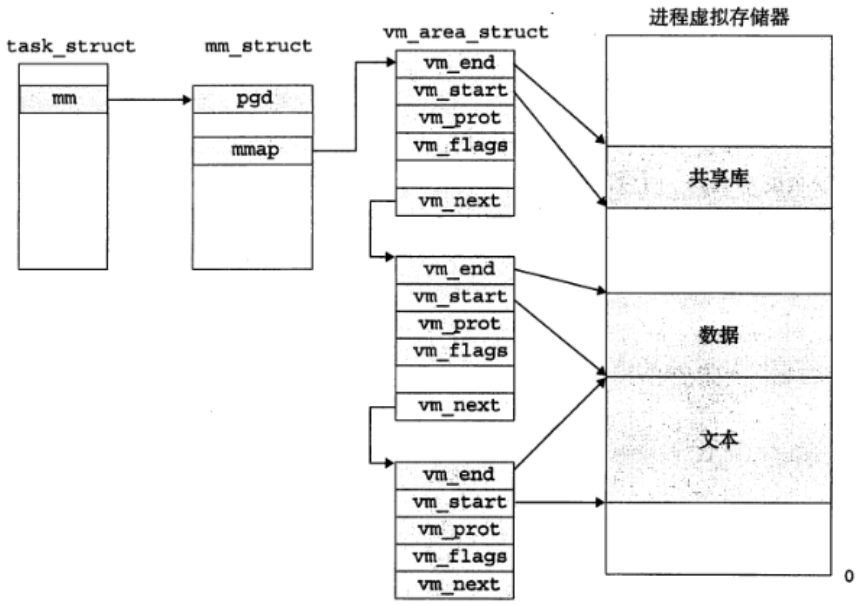
② Linux虚拟存储系统



一个单独的Linux系统进程虚拟存储主要分为：内核虚拟存储器和进程虚拟存储器。

我们主要来讲一下内核虚拟存储器：由下往上是内核的代码和数据结构，是每个进程共享的数据结构和代码；再往上是一组连续的虚拟页面映射到相应的物理页面的物理存储器，大小同主存一样大，提供很方便访问物理页面的任何位置。最后是每个进程不同的是页表、task（mm）、内核栈等。

虚拟存储器区域：



区域就是我们通常说的段，text、data、bss都是不同的区域，这些区域是被分为连续的片。每个虚拟页面都在不同的段中，不属于某个段的虚拟页面是不存在的，且不能被使用。我们来看看内核中的一个task数据结构（mm）：

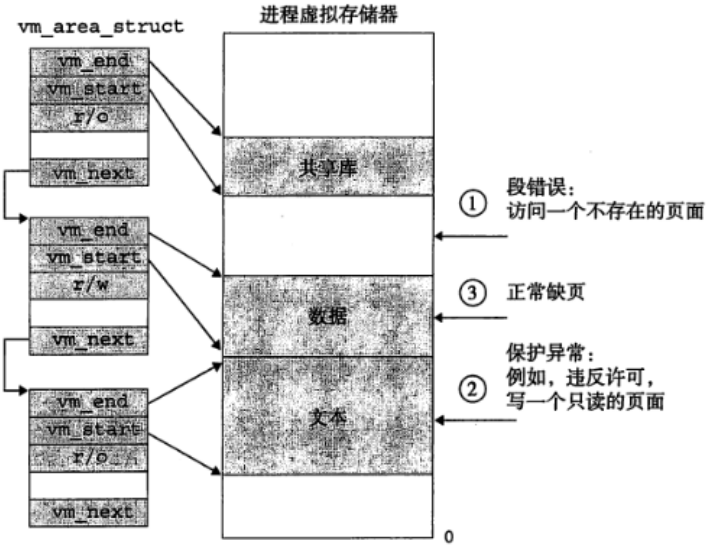
task_struct是位于内核虚拟存储器中对于每个进程的都不同的内核数据结构，包含运行该进程所需要的基本信息（PID、可执行文件名称、程序计数器等）。这个结构中有一个mm字段，指向的是mm_struct中的pgd和mmap，其中pgd是一级页表的基地址，mmap指向的是一个vm_area_structs的链表，每个该链表中的一个元素描述的是当前虚拟地址空间的一个段（text、data、bss等），当内核运行该进程的时候CR3寄存器就被放入了pgd。

Linux缺页异常处理：

我们将了一些存储器区域划分的基础知识，并且介绍说mmap指向的是一个链表，这个链表中的每个元素都指向该进程的相应的段，其中vm_strat是段开始的地方，vm_end是段结束的地方。

1> 访问地址是否合法：缺页处理程序只需要将这个地址A与vm_area_struct链表中的每个元素的start和end数据比较，如果都没有的话，表示该地址不在相应的段中。就是一个段错误。

2> 保护异常：vm_area_struct中的vm_prot结构是包含了所有页面的读写权限，所以当对只有读权限的文本内容写入数据的时候，就会引发保护异常。



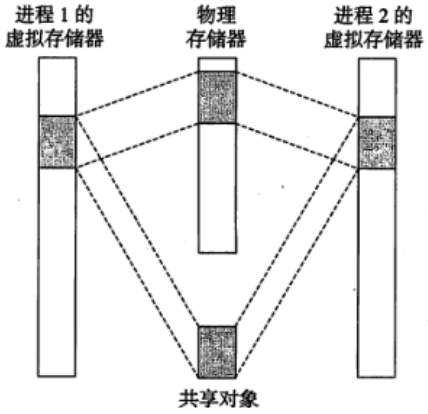
3> 最后，正常缺页。也就是相应的页面不在物理内存的时候，缺页程序就会锁定一个牺牲页面，将它的内容与实际需要的内容交换过来，当缺页程序返回的时候就可以正常的访问了。

1.7 存储器映射

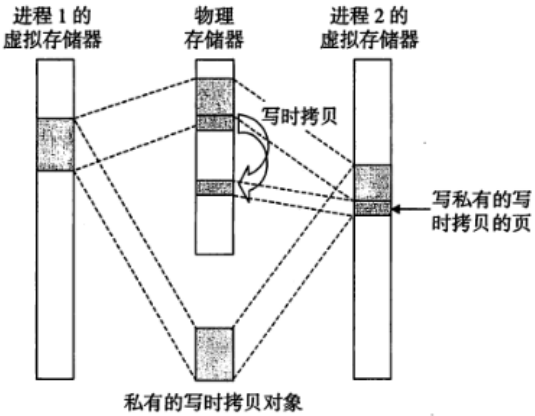
存储器映射是通过将磁盘上的一个文件与虚拟存储器中的一个区域关联起来的过程。

① 理解共享对象

一个对象被映射到虚拟存储器的一个区域，这个区域要么是共享对象，要么是私有对象。如果一个进程A将一个共享对象映射X到了它的虚拟存储器中，那么对于也把这个共享对象X映射了的其他进程而言，进程A对共享对象X的任何读写操作都是可见的。下图是进程1和进程2映射了共享区域的图例：



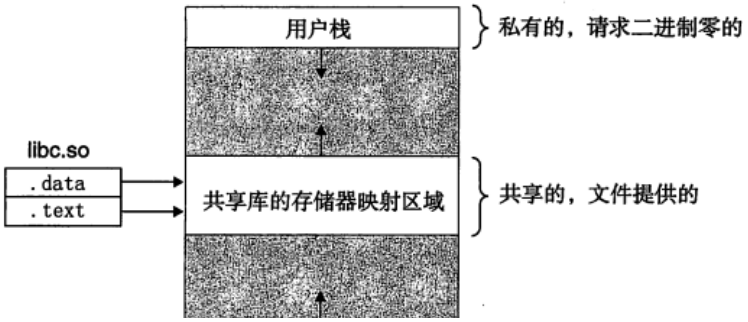
私有区域：即使是私有区域在物理存储器上也是同一个区域，如下图进程1和进程2所映射的私有对象在物理存储器上只是一份拷贝。



每个对象都有唯一的一个文件名，在进程1的虚拟存储器中已经完成了私有对象到存储器的映射，进程2如果要映射这个区域只需要将页表条目指向已经映射好的物理存储器位置就行了。如上图所示，进程1和2将一个私有对象映射到了物理存储器的一个区域并共享这个私有对象。这个对象会被标记为只读，当其中一个进程2确实需要写这个区域的时候，就会引发一个保护故障，内核会在物理存储器中创建这个私有对象的一个拷贝，称为写时拷贝，更新页面条目使得进程1指向这个新的条目。然后把老对象修改为可写权限。这样当保护故障程序返回的时候，CPU从新执行写的操作就不会出错了。

② 理解fork函数如何创建独立的虚拟地址空间

当当前进程调用fork函数的时候，内核为新进程创建各种数据结构，并分配PID。为了给新进程创建一个虚拟存储器，它创建的当前进程的mm_struct、区域结构和页表的一个拷贝，内核为两个进程的每个页表标记为只读，并将诶个区域标记为私有的写时拷贝。这样当fork函数返回的时候，新进程的虚拟存储器和当前进程的虚拟存储器刚好相同。任何一个进程进行写操作的时候，才会创建新的页面。



③ 理解execve函数实际上如何加载和执行程序

1> 删除已存在的用户区域；

2> 映射私有区域

所有的.text、.data、.bss区域都是新创建的，这些区域是私有的、写时拷贝。.bss是匿名文件区域，初始化为二进制0，栈、堆也都是初始化为0.

3> 映射共享区域；

这些共享区域是动态链接到程序然后映射到虚拟地址空间的共享区域。

4> 设置程序计数器。

最后一步就是设置当前进程的上下文计数器，并指向.text入口

④ 使用mmap函数创建新的存储器映射

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

void  *mmap(void *start, size_t length, int prot, int flags,
            int fd, off_t offset);
                Returns: pointer to mapped area if OK, MAP_FAILED (−1) on error
```

函数原型如下：

> 说明：
>
> start：从地址start开始处创建，通常为NULL；length：连续对象的大小；
>
> port：访问权限（PROT_EXEC\PROT_READ\PROT_WRITE\PROT_NONE）；
>
> flags：被映射对象的位（MAP_ANOE\MAP_PRIVATE\MAP_SHARED）；
>
> fd: 指定的磁盘文件；offset：距离磁盘文件偏移的位置处开始；
>
> 返回值：调用成功，返回新区域的地址。

（注：可以使用munmap删除相应的虚拟存储器区域）

1.8 动态存储分配

动态存储器分配指的是在程序运行的时候分配额外的存储空间，分配器维护着虚拟存储器中的堆实现这种分配。

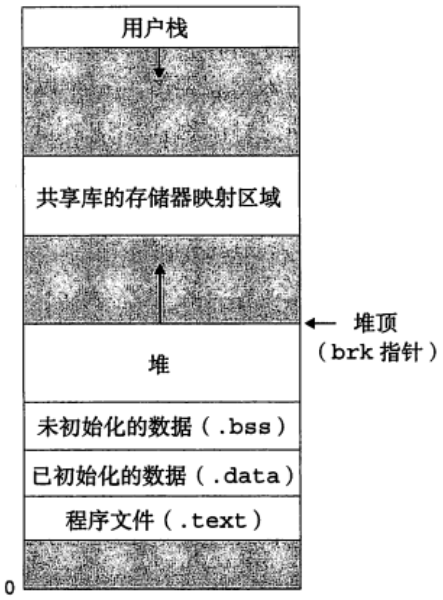


图 9-33　堆

堆是紧跟着.bss段，并向上增长，内核维护着一个brk指针，指向堆的顶部。任何一个堆中的块要么是已分配的要么是空闲的。分配的方式分为两种：显式和隐式，我们接下来主要讲一下显示分配和实现一个分配器的基础知识，隐式分配指的其实是分配器回收空间，这个在分配器基础知识中有所讲解，就不再另外提出了：

① 显式分配：程序调用malloc和free函数

经常直到我们的程序运行的时候，我们才知道某些数据结构的大小。这时候就必须显式的分配相应的存储空间。如下图所示：

```
int main()
{
   int *array, i, n;

   scanf("%d", &n);
   array = (int *)Malloc(n * sizeof(int));
   for (i = 0; i < n; i++)
      scanf("%d", &array[i]);
   exit(0);
}
```

```
#include <stdlib.h>

void *malloc(size_t size);
```

使用malloc函数以具体的输入内容分配相应大小的存储空间，函数原型如下：

如果想要初始化存储器为0，可以使用calloc函数。想要改变已分配的大小可以使用realloc函数

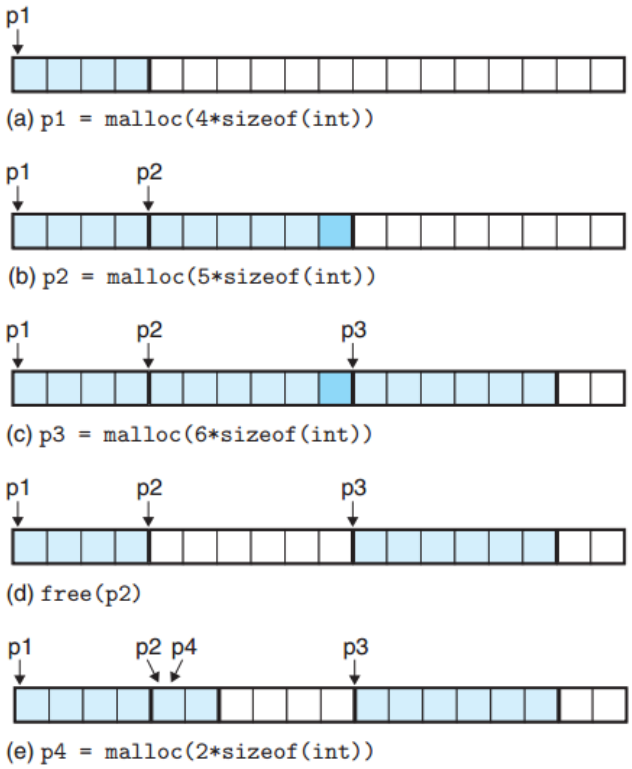
释放是通过调用free函数来实现的：

```
#include <stdlib.h>

void free(void *ptr);
```

ptr是指向一个已分配空间的起始位置

我们来看一个分配实例：



> （a）请求一个4字大小的块，malloc将分配好的空间的首地址返回给p1；
>
> （b）请求一个5字大小的块，由于使用的双字对其，所以填充了一个空闲块；
>
> （c）请求一个6字大小的块，返回给p3；
>
> （d）释放p2，调用后p2仍然指向原来的位置；
>
> （e）请求一个2字大小的块，在已经释放的p2处优先分配，然后返回指针p4

② 分配器基础知识

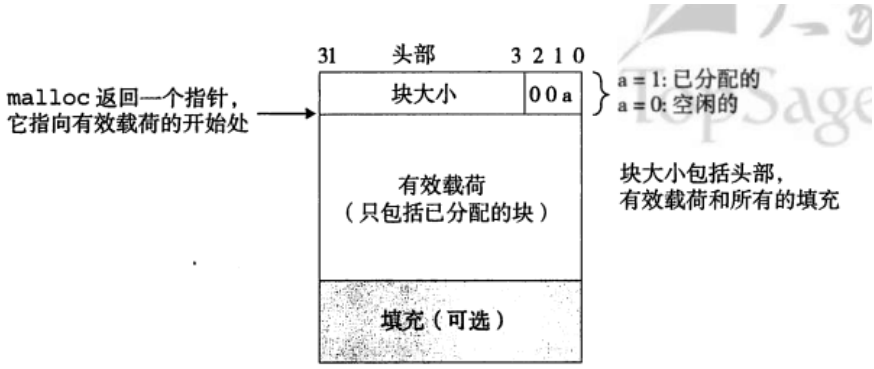
分配器的目标主要是找到吞吐量和利用率的契合点，那么为什么需要隐式的分配，因为碎片的产生会降低存储空间的利用率

碎片：内部和外部

1>内部碎片：我们上面讲到的（b）的情况，分配了一个额外的空闲块，实现双字对其；

2>外部碎片：(e)中如果请求7字大小的块，即使存储空间有这么大，还是不行

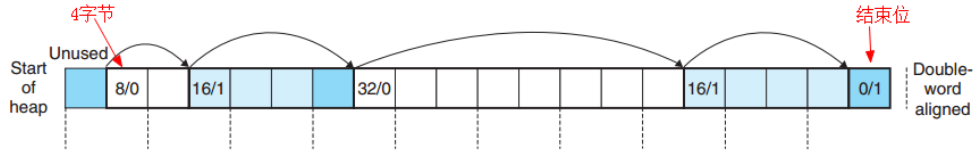
当然，还有许多问题要思考，诸如：空闲块如何组织、如何分配新的块、怎么分割和合并块，这些技术都要求我们提供一种新的数据结构



隐式空闲链表：

这样的一种结构，主要是由三部分组成：头部、有效载荷、填充（可选）；

头部：是由块大小+标志位（a已分配/f空闲）；**有效载荷：**实际的数据



应用举例：

这个链表（大小/是否分配）是通过头部中的大小字段 隐含连接着的（头部+大小=下一块位置），分配器可以遍历所有的块，在遇到结束位（0/1）处停止。即使是要求分配一个数据块，也要有（8/0）一个头部，两个字来完成。

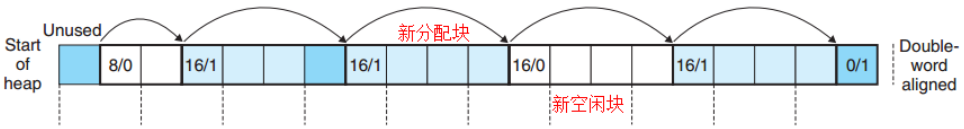
简单的放置策略：

1> 首次适配：从头搜索，遇到第一个合适的块就停止；

2> 下次适配：从头搜索，遇到下一个合适的块停止；

3> 最佳适配：全部搜索，选择合适的块停止。

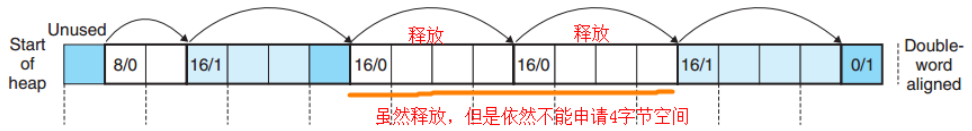
分割空闲块：



适配到合适的空闲块，分配器将空闲块分割成两个部分，一个是分配块，一个是新的空闲块：

增加堆的空间：通过调用sbrk函数，申请额外的存储器空间，插入到空闲链表中

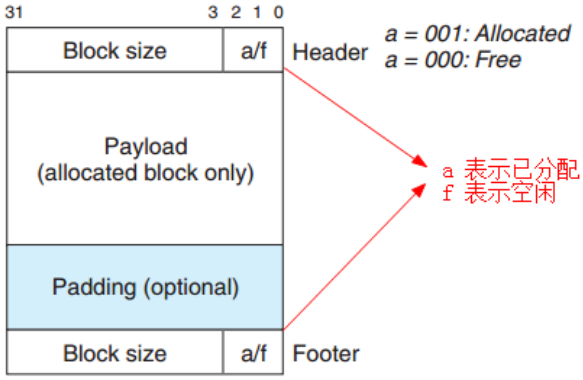
合并：



1> 为什么要合并：处理假碎片现象

如上所示，虽然释放了两个3字节大小的数据空间，而且空闲的空间相邻，但是就是无法再分配4字节的空间了，这时候就需要进行一般合并：合并的策略是立即合并和推迟合并，我们可能不立即推迟合并，如果有空间直接合并不好吗？有时候的确还真不好，如果我们马上合并上图的空间后又申请3字节的块，那么就会开始分割，释放以后立即合并的话，又将是一个合并分割的过程，这样的话推迟合并就有好处了。需要的时候再合并，就不会产生抖动了。

2> 怎样合并：带边界标记

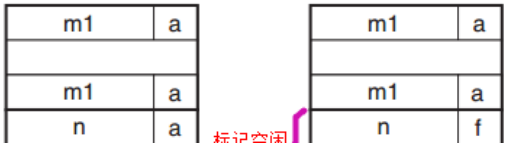


我们需要从新审视一下我们的隐式链表数据结构，加入新的边界标记形成如下结构：

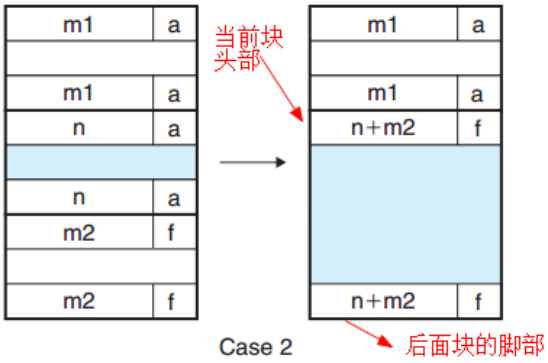
在链表的底部加入头部同样的格式，用a表示已分配、f表示空闲

我们列举一下可能的所有情况：

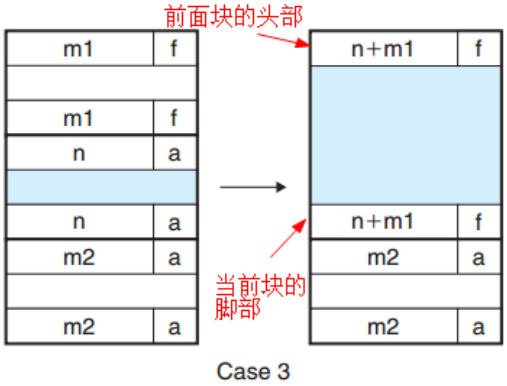
说明：



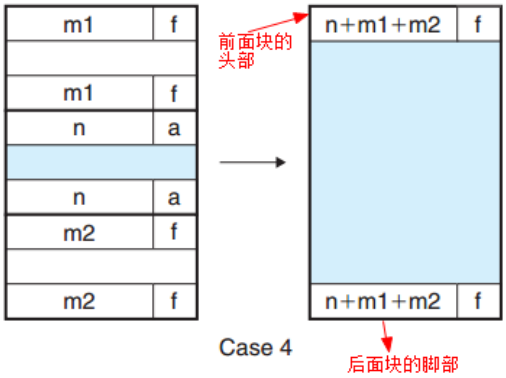
（a）：在合并的时候，由于前后都是已分配不执行合并，只是把当前块标记位空闲：



（b）：后面的块是空闲的，当前块与后面的块合并，用新的块的大小（当前块大小+后面块大小），更新当前块的头部和后面块的脚部



（c）：前面块是空闲，前面块与当前块合并，用新的块的大小（当前块的大小+前面块的大小），更新前面块的头部和当前块的脚部



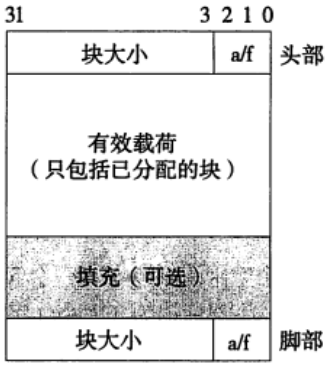
（d）：三个块都是空闲，3个块的大小来更新前面块的头部和后面块的脚部

当前已分配的头部

注意：当（c）和（d）两种情况，前面的块是空闲的，才需要用到当前块的脚部。（a）不需要更新，（b）更新的是后面块的脚部+块大小。如果我们把前面的块的位存放在当前块头部未使用多出来的低位中，那么已分配的块就不需要脚部了。（当然空闲块仍然需要脚部）

③ 综合：实现一个简单的分配器

我们将实现的是一个基于隐式空闲链表，立即边界标记合并方式的简单分配器。数据的结构如下：



最小的块大小为16字节，必须包含：头部（8字节）+脚部（8字节）；

隐式的空闲链表具有如下恒定的形式：

其中首字（对齐）是不使用的填充块，紧跟着的是一个特殊的序言块，由一个头部和脚部组成，序言块在初始化的时候创建，永不释放。中间就是由malloc创建的普通块，最后是一个特殊的结尾块，序言和结尾块都是为了消除合并边界条件的技巧。（heap_listp总是指向序言块）

接下来的内容，我们直接上与malloc和free有关的函数源码，用到什么知识的时候再做补充：为了和系统的malloc和free函数区分，我们起名为（mm_malloc和mm_free）：

```c
int mm_init(void)
{
    // 创建一个空的堆，heap_listp指向序言块
    if(heap_listp = mem_sbrk(4*WSIZE) == (void *)-1)
        return -1;
    PUT(heap_listp, 0); // 首字节
    PUT(heap_listp +(1*WSIZE), PACK(DSIZE, 1)); // 序言块头
    PUT(heap_listp+(2*WSIZE), PACK(DSIZE,1));   // 序言块脚
    PUT(heap_listp+(3*WSIZE), PACK(0,1));       // 结尾块
    heap_listp += (2*WSIZE);                    // 指向第一个块
    // 将堆扩展CHUNKSIZE字节，创建空闲块
    if(extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;

    return 0;

}
```

这里我们要对extend_heap函数进行说明，在扩展堆的时候必须要调用mm_init函数进行初始化，创建一个空的链表。初始化mm_init函数如下：

```c
static void *extend_heap(size_t words)
{
    char *bp;          // 指针
    size_t size;       // 大小

    // 分配偶数字节
    size = (words %2 ) ? (words+1) * WSIZE : words *WSIZE;
    if((long)(bp = mem_sbrk(size))== -1) // 请求额外的堆空间
        return NULL;
    // 填充相应的字节
    PUT(HDRP(bp), PACK(size, 0));   // 新空闲块的头部
    PUT(FTRP(bp), PACK(size, 0));   // 新空闲块的脚部
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0,1));  // 新结尾块的头部

    // 如果前面的块是空的就进行合并
    return coalesce(bp);

}
```

新的扩展空闲空间返回的是紧跟着老结尾块的后面

具体的扩展函数如下，当堆初始化或者mm_malloc函数不能匹配的时候就会进行扩展：

```c
void mm_free(void *bp)
{
    // 获取释放空间大小
    size_t size = GET_SIZE(HDRP(bp));
    // 开始释放
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    // 合并的4种方法
    coalesce(bp);

}
```

分配我们讲完了，下面来看看mm_free函数和合并空闲块函数coalesce函数，合并的4种情况：

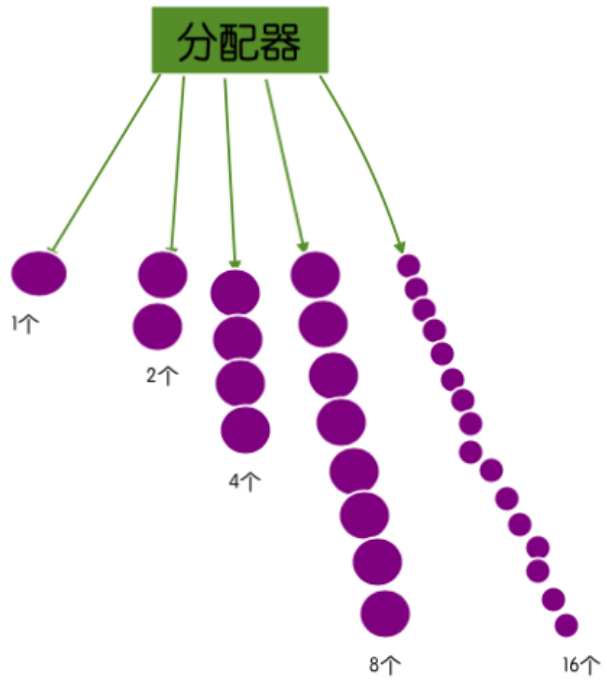
```c
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))); // 前块
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp))); // 后块
    size_t size = GET_SIZE(HDRP(bp));                   // 大小

    // 情况1：直接释放，前块和后块都是已分配
    if(prev_alloc && next_alloc)
    {
        return bp;
    }
    // 情况2：前面已分配后面未分配
    else if (prev_alloc && !next_alloc)
    {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    // 情况3：前面未分配后面已分配
    else if(!prev_alloc && next_alloc)
    {
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0)));
        bp = PREV_BLKP(bp);
    }
    // 情况4：前后都位分配
    else
    {
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    return bp;

}
```

④ 分离的空闲链表



我们使用单向的空闲链表分配时间并没有改善，我们现在来看看比较流行的分离存储方法，在一个分离存储的系统中，分配器维护着一个空闲链表数组，每个空闲链表是一些大小不同的类。我们可以按照2的幂来划分，比如1，2，4，8，16，32.....等等。

像上图那样，每个大小类都是2的幂，按照升序排列就是伙伴系统。我们要分配一个9字节的空间，就需要从前往后依次寻找，找到了第5个空闲链表中的空间足够就分割它，将不需要的插入到空闲链表中去。如果找不到合适的，比如需要17个字的空间，就向操作系统申请额外的堆存储器。

申请：如果我们要在16字节的空间中分配一个4字节大小的空间，就会首先将这个16字节的总空间分割成两个8字节大小的空间，其中空闲的那部分（左边）叫做伙伴，被放置到空闲链表中。我们发现8字节的空间依然大于我们要分配的空间，就再一次将8字节的空间分割成两部分，每个4字节，刚好完成分割，这时候8字节中的左边部分也就是伙伴，被放置到空闲链表中。

释放：需要释放4字节空间的时候，会与其伙伴进行空闲合并，形成一个8字节大小的空闲空间，继续发现另外的8字节伙伴也是空闲的，继续合并。直到遇到的伙伴已经被分配了才停止。

⑤ 垃圾收集

垃圾收集是一种很有用的方法，当使用了malloc分配了空间却忘记了释放，就会造成内存的极大浪费。垃圾收集就是使用特殊的方法，定期回收这部分不使用或者无效的空间。

当然收集的方法分为很多种，我们只讲一下【标记清除算法】：

垃圾收集器将存储器视为一张有向的图，根节点保存在寄存器、栈变量或者虚拟存储器的全局变量中，子节点在堆中，每个子节点对于一个已分配的块。白色为可到达，蓝色为不可到达应该被回收的区域。

垃圾回收期就是维护着这种可达图，释放其中不可达的节点，返回给空闲链表。

像c和c++这类语音不能维持可达图的精确表示，有些不可达的节点可能会被错误的标识为可

达，它的垃圾收集器就是一个保守的垃圾收集器。加入到malloc包中就形成这样的形式：

当我们需要空间的时候，调用了malloc函数，如果找不到合适的空间就会启用垃圾收集器，希望回收一部分可利用的空间，垃圾收集器将代替程序执行free函数，释放的空间返回以后重新调用malloc，如果还是失败才从操作系统申请额外的存储空间。

如何标记：

标记前的状态是这样的：

淡蓝色的块是已分配的头部，6个块都是未标记的。其中根节点指向块4，4的又分别指向了3和6.其中3又指向了1，这就形成了一个有向的链表，其中只有2和5不能到达。

这时候我们使用标记函数，循环遍历进行标记：

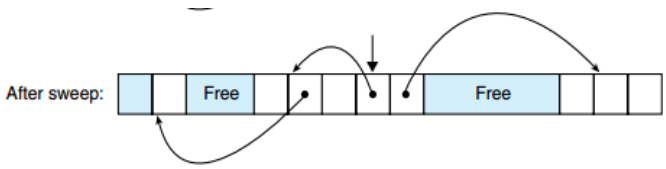
标记完成以后，形成如下所示：

由于块1、3、4、6是可达的，所以都被标记了。2和5无法标记是垃圾

sweep函数释放所有的未标记的块：

(b) sweep function

```
void sweep(ptr b, ptr end) {
   while (b < end) {
```



清除块2和块5以后是这样的：

我们之所以说c和c++是保守的垃圾收集器，是因为在标记阶段的isPtr函数识别并不准确，c不知道输入的参数是否是一个指针，也不知道指向的是否是一个有效载荷的位置。（**这里需要多读读，还有点儿不懂**）

1.9 c程序中常见的10大存储器相关错误

存储器的错误总是令人沮丧的，特别是在运行了一段时间之后才显示出来，就特别特别的烦人了，我们列举一些常见的错误，仅供参考：

① 间接引用坏指针：将本来的地址引用写成了内容scanf("%d", &val)写成scanf("%d", val)

② 读未初始化的存储器：

在堆中申请了一块空间：int *y = (int *)Malloc(n * sizeof(int));由于堆中的空间是未被初始化的，下面的使用就会出错：y[i] += A[i][j] * x[j];推荐使用calloc函数

```
void bufoverflow()
{
    char buf[64];

    gets(buf); /* Here is the stack buffer overflow bug */
    return;
}
```

③ 允许栈缓冲溢出：

推荐使用fgets函数

④ 假设指针和它指向的对象是相同大小的

使用int **A = (int **)Malloc(n * sizeof(int));本来是想创建的一个int *的数组，但是sizeof上面用到的确实int

⑤ 错位

```
/* Create an nxm array */
int **makeArray2(int n, int m)
{
    int i;
    int **A = (int **)Malloc(n * sizeof(int *));
```

申请了n个空间，却要访问n+1处位置

⑥ 引用指针而不是它指向的对象

size--; / This should be (*size)-- */

其中，--和*有相同的优先级，由于这是右结合。所以先--再*，就出错了。

⑦ 误解指针运算

p += sizeof(int); /* Should be p++ */

指针p++就会指向下一个位置，+= int的大小的话，就跳了几个数据了

⑧ 引用不存在的变量

```
int *stackref ()
{
    int val;

    return &val;
}
```

本地变量在栈中创建，函数结束以后就已经不存在了。

```
1   int *heapref(int n, int m)
2   {
3       int i;
4       int *x, *y;
5
6       x = (int *)Malloc(n * sizeof(int));
7
8       /* ... */   /* Other calls to malloc and free go here */
9
10      free(x);
11
12      y = (int *)Malloc(m * sizeof(int));
13      for (i = 0; i < m; i++)
14          y[i] = x[i]++;  /* Oops! x[i] is a word in a free block */
15
16      return y;
17  }
```

⑨ 引用已经释放了的块中的数据

简书 首页 下载APP 搜索 Aa beta 登录 注册

```
void leak(int n)
{
    int *x = (int *)Malloc(n * sizeof(int));

    return;  /* x is garbage at this point */
```

10 不释放引起 泄漏

21赞

如果经常调用leak 又不释放的话，内存就会充满垃圾。

赏

2017年04月28日23:59:58 完

赞赏

21人点赞 > 《深入理解计算机系统》

"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下

唐鱼的学习探索 如果我像一般人一样读那么多书，我就跟他们一样愚蠢了。
总资产29 (约2.85元) 共写了10.4W字 获得530个赞 共463个粉丝 关注

ExpressVPN™- Award-Winning VPN
With a VPN You Can Surf the Internet with No Censorship. Blazing-Fast Speeds!

写下你的评论...

全部评论 1 只看作者 按时间倒序 按时间正序

趁年轻奋斗吧
2楼 04.13 00:03

赞！

赞 回复

被以下专题收入，发现更多相似内容

写下你的评论... 评论1 赞21 ···

推荐阅读

三面字节跳动被虐得“体无完肤”，15天读完这份pdf，终拿下美团研发
阅读 79,546

对于二本渣渣来说，面试阿里P6也太难了！（两年crud经验，已拿
阅读 45,124

面试官再问你 HashMap 底层原理，就把这篇文章甩给他看
阅读 5,103

离开菜鸟&新的面试体验
阅读 9,346

败给“MySQL”的第33天，我重振旗鼓，四面拿下阿里淘系offer
阅读 6,283

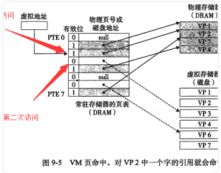
推荐阅读

更多精彩内容 >

《深入理解计算机系统》-虚拟存储器

你是否疑惑过为什么两个进程可以拥有同一个地址，明明某个地址处的物理内存只有一块啊。这其实是系统提供的一种对主存的一...

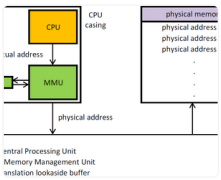
gatsby_dhn 阅读 1,302 评论 1 赞 4



虚拟内存的那点事儿

概述 我们都知道一个进程是与其他进程共享CPU和内存资源的。正因如此，操作系统需要有一套完善的内存管理机制才能防止...

SylvanasSun 阅读 2,572 评论 0 赞 22



东北大学软件学院操作系统复习考点（附概念题）

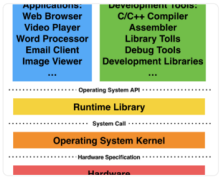
word直接复制来了，格式就不改了。至于这门课怎么复习，只要平时实验都认真完成、报告认真写，平时分都很高；考试的话...

Jozhn 阅读 1,694 评论 0 赞 5

iOS 程序员的自我修养 — 读《程序员的自我修养-链接、装载...

2016年国庆假期终于把此书过完，整理笔记和体会于此。 关于书名 书名源于俄罗斯的演员斯坦尼斯拉夫斯基创作的《演员...

李剑飞的简书 阅读 4,247 评论 1 赞 53



往事如烟 逝去吧

往事不能回忆 甜蜜也好 痛苦也罢 回忆一次伤一次.

狍子哥哥 阅读 23 评论 0 赞 1