

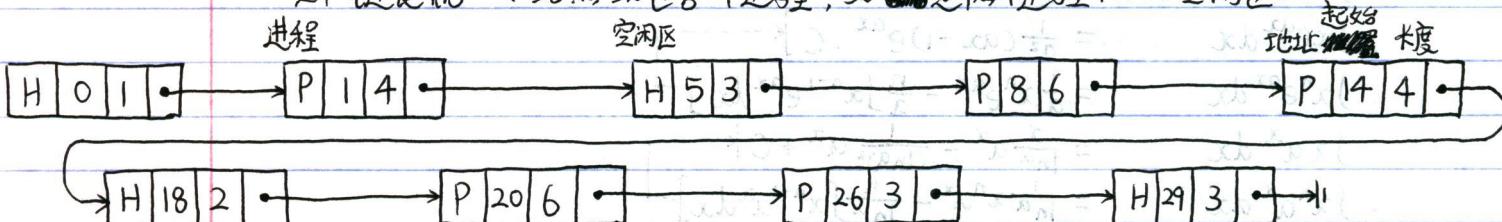
Operating System - P38

2019-2020

空闲区链表

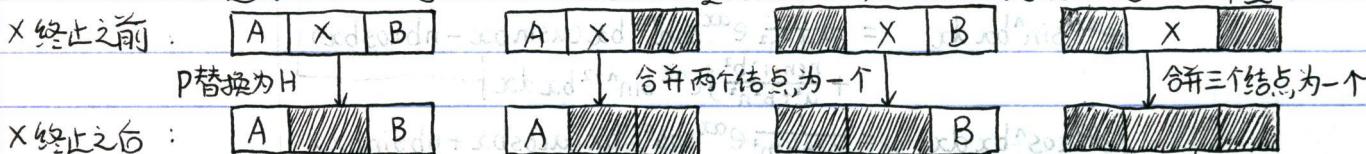
为了记录内存的使用情况，维护一个记录已分配内存段和空闲内存段的链表

其中链表的一个结点，或包含一个进程，或包含两个进程间的空闲区



链表结点，包含：空闲区(H)或进程(P)的指示标志，起始地址，长度和指向下一结点的指针

当段链表是按地址排序的，其优点是当进程终止时或被换出时链表的更新非常直接



注意：此处所用的单向链表，而若要实际使用的话，可使用双向链表 (double linked list)

为创建的进程（或从磁盘换入的已存在的进程）分配内存的算法

首次适配(first fit)，存储管理器沿着段链表搜索，直到找到一个足够大的空闲区

除非空闲区大小与分配大小一致，否则将空闲区分成两部分，

一部分分配给进程，另一部分形成新的空闲区

下次适配(next fit) 相比于首次适配算法，下次适配算法每次找到合适的空闲区时都记录当时的位置

在下次寻找空闲区时从上次结束的位置开始搜索

但仿真程序证明下次适配算法略低于首次适配

最佳适配(best fit)，搜索整个链表，并找出能够容纳进程的最小的空闲区

最佳适配算法试图找出最接近实际需要的空闲区，以最好地匹配需求

首先由于每次都要搜索整个链表，则比首次适配算法更慢

另外最佳适配会产生大量无用的小空闲区，而首次适配会生成更大的空闲区

注意：如果为进程和空闲区维护各自独立的链表，则可以提高算法的速度

但是代价是增加系统的复杂度

最差适配(worst fit) 相比于最佳适配算法，最差适配总是分配最大的可用空闲区

但仿真程序表明最差适配也不是好的算法

注意：如果使用不同链表，则可以按照大小对空闲区链表排序，以提高最佳适配算法的速度

此时首次适配与最佳适配算法一样快，而下次匹配则失去意义，因为链表顺序和地址顺序不相关

注意：可以用空闲区本身作为链表节点，即空闲区的头两个字节分别存储空闲区大小和指向下一个空闲区的地址

快速适配(quick fit)，为常用大小的空闲区维护单独的链表，在寻找指定大小的空闲区时非常快速

但缺点是进程终止和换出时，寻找相邻块并合并非常费时

但如果不行则会分裂出大量无法利用的小空闲区

Operating

System - P39

膨胀

(bloatware). 虽然存储器容量增长快速，但是软件大小增长更快

其结果是，需要运行的程序往往大到内存无法容纳，且必须需要系统支持同时运行多个程序

覆盖

(overlay). 20世纪60年代的解决方案，即把程序分割成许多片段，并有覆盖管理模块

先将覆盖管理模块装入内存，并装入运行覆盖0，运行完成后装入运行覆盖1，以此类推
覆盖块存放在磁盘，需要时由OS动态换入换出，有时允许多个覆盖块同时在内存中
有空间时覆盖块在内存中顺序装入，没有空间时则占用之前的覆盖块

缺点是由于需要完成实际的覆盖块换入换出操作，程序员必须将程序分割成多个片段
这个过程非常费时和枯燥的，并且容易出错

虚拟内存
页

(virtual memory)，其基本思想是每个程序拥有自己的地址空间，并被分割成多块
(page)，或称页面，每页有连续的地址范围，
页被映射到物理内存

但并非所有页都必须在内存中才能运行程序，即无需将程序全部装入内存

当引用到在物理内存的地址空间时，由硬件立刻执行必要的映射

当引用到不在物理内存的地址空间时，由OS将缺失部分装入内存并重新执行失败指令

虚拟内存适合于多道程序设计系统，多个程序的片段都被保存在内存中

当一个程序等待缺失部分装入，可以将CPU交给另一程序

分页

(paging)，一种应用于大部分虚拟内存系统中的技术

程序引用的内存地址，可以通过索引，基址寄存器，段寄存器或其他方式生成

虚拟地址 (virtual address)，指这些由程序生成的地址，构成了虚拟地址空间 (virtual address space)

在没有虚拟内存的计算机，OS直接将虚拟地址送到内存总线，读写操作相同的物理地址

内存管理单元 (Memory Management Unit, MMU)，在使用虚拟内存时，将虚拟地址映射为物理内存地址

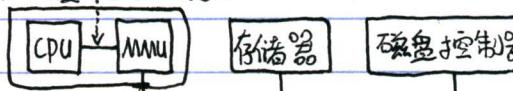
页框

(page frame)，物理内存中与虚拟地址空间中按固定大小划分的页 (page) 对应的单元

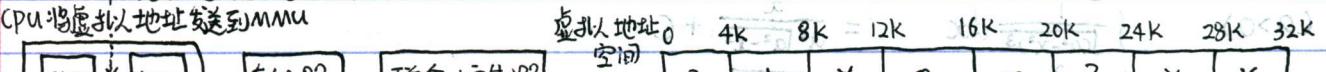
通常页面与页框具有同样的大小，RAM与磁盘间的交换都是以页面为单位进行的

CPU将虚拟地址发送到MMU

CPU包



总线

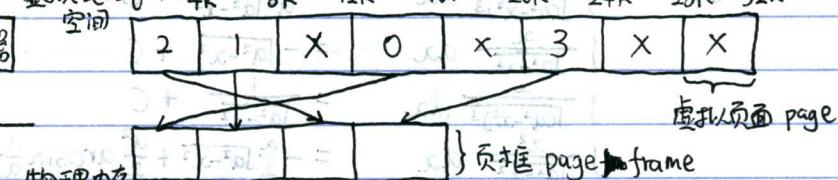


MMU将物理地址通过总线送给存储器

注意：将MMU作为CPU的一部分是因为其逻辑

但实际上是由独立于CPU的芯片

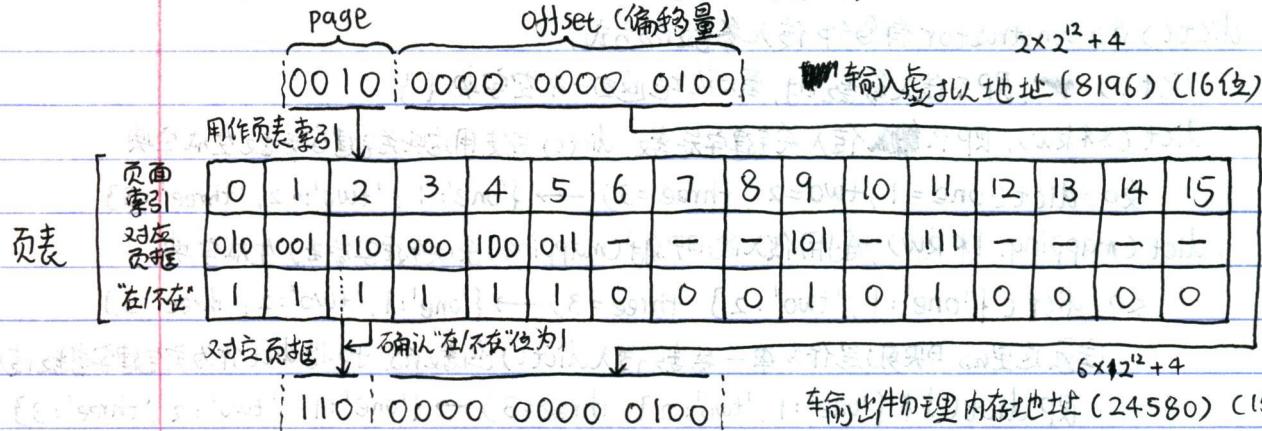
另外：实际上内存存储器并不知晓MMU的存在



Operating

System - P40

"在/不在" (present/absent bit), 在实际写入件中记录页面在内存中的实际存在情况



缺页中断 (page fault) 或称缺页错误, 即当 MMU 查询页表索引得到“在/不在”位为0时, 即表示该页面没有映射

此时引起一个操作系统陷阱, 使 CPU 陷入操作系统

OS会选择放弃一个已映射的页框, 将其内容写回磁盘,

然后从磁盘将需要访问的页面读入页框, 并修改页表中的映射, 再重新启动引起陷阱的指令

页表 (page table) 用于虚地址页面映射为页框, 或者说将输入的虚地址转化为物理地址输出

虚地址被分为虚地址页号 (高位) 和偏移量 (低位), 如 16 位虚地址分为页号 (4 位) 和偏移量 (12 位)

页表以虚地址页号作为索引, 找到对应的页表项, 如果“在/不在”位为0 则引起缺页中断

如果为1 则取页框号 (3位) 拼接到偏移量 (12位) 的高位端, 形成内存物理地址 (15位)

从数学角度说, 页表是一个从虚地址页号到物理页框号的函数: page \rightarrow frame $\cup \{null\}$

可以把虚地址中的虚地址页域替换成页框域, 从而形成物理地址

不完全虚地址化 即实现页表过程中不涉及虚地址机, 从而设计结构相对简单, 虚地址机中页面调度和虚地址内存相当复杂

由于每个虚地址机都需要自己的虚地址页表, 页表组只可能包括影子页表和嵌套页表

结构

页表项

页框号

注意在这种模型中

高速缓存禁止位

修改位

“在/不在”位

页表索引是独立于页表项

访问位

保护位

的 部分

页表项常用的大小为 32 位, 其中最重要的域是页框号, 由于这是映射的目的

其次是“在/不在”位, 为1 表示该表项有效, 为0 则表示虚地址不在内存中, 进而引起缺页中断

保护 (protection) 位指出一个页允许什么类型的访问, 基本形式为 4 位, 为0 表示读写, 为1 表示只读

改进的做法是包含三位, 分别对应: 读 (read), 写 (write), 执行 (execute) 该页面

注意这种形式与文件管理系统的保护位设计逻辑一致, 如 UNIX 系统的 rwx 位

Operating System - P41

Page - Cache

修改 (modified) 位，用于记录页面的使用情况，在写入页时由硬件自动设置修改位

特别在 OS 重新分配页框时，如果页面修改过 (dirty)，则必须写回磁盘以保存修改

如果没有修改过 (clean)，则只需简单丢弃即可，因为留在磁盘的副本仍然有效
所以修改位也称为脏位 (dirty bit)

访问 (referenced) 位，也是用于记录页面使用情况，不论读写，OS 都会在访问该页时设置访问位

特别在 OS 发生缺页中断需要选择被淘汰的页面时。

不再使用 (最近未被使用) 的页面要比正在使用的页面更适合被淘汰

禁止高速缓存位 用于对于映射到设备寄存器而非常规内存的页面，禁止该页面被高速缓存

特别在 OS 循环等待 I/O 设备对刚发出的命令作出响应时

保证硬件是不断从设备中读取数据而不是访问一个旧的被高速缓存的副本

注意：有独立的 I/O 空间而不使用内存映射 I/O 的机器不需要这一位

由于具有独立的 I/O 空间，CPU 无需通过多层次存储体系寻找 I/O 空间的数据。

即不存在因为高速缓存而导致新数据无法覆盖旧数据的情况。

注意：页表只保存把虚拟地址转换为物理地址的硬件所需的信息

即如果某个页面不在内存中，用于保存该页的磁盘地址不是页表的组成部分

OS 在处理缺页中断时会将磁盘地址等信息保存在 OS 内的软件表格

注意：虚拟内存本质上是用来创造一个新的抽象概念，即地址空间

加速分区过程 虚拟地址到物理地址的映射必须非常快，每条指令进行更多次页表访问是必要的

由于每次访问内存都需要进行虚拟地址到物理地址的映射

所有的指令最终都必须来自内存，且很多指令也会访问内存中的操作数

如果虚拟地址空间很大，页表也会很大，且每个进程都需要单独的页表 (虚拟地址空间)

一种简单方案是使用由“快速硬件寄存器”阵列组成的单一页表，每一页表项对应一个虚拟页面
启动进程时，OS 把保存在内存中的进程页表的副本载入寄存器，则运行中不必再为页表访问内存

优点是设计和实现简单并且在映射过程中不需要访问内存

缺点是页表很大代价高昂，且每一次上下文切换都必须装载整个页表

另一种方案是，整个页表都在内存中，则所需的硬件仅为一个指向页表起始位置的寄存器

优点是在上下文切换时，进行“虚拟地址到物理地址映射”只需装入一个寄存器

缺点是执行每条指令时，都需要内存访问来完成页表项的读入，速度很慢

Operating

System - P42

转换检测缓冲区(Translation Lookaside Buffer, TLB), 又称相联存储器(associative memory)或快表

通常为设置在MMU中的小型硬件设备, 包含少量表项

用于解决采用分页机制时, 只访问一次内存的指令被迫多次访问内存而使性能下降的问题

TLB 将虚拟地址直接映射到物理地址 有效位 虚拟页面号 修改位 保护位 负责权

而不必再访问页表 1 140 1 RW- 31

每个表项记录一个页面的相关信息 1 20 0 R-X 38

包括虚拟页面号, 页面的修改位 1 130 1 RW- 29

保护码(读/写/执行权限) 1 129 1 RW- 62

该页所对应的物理页框 1 19 0 R-X 50

虚拟页号(不必放在页表中) 1 21 0 R-X 45

有效位(是否正在使用) 1 860 1 RW- 14

基于观察: 大多数程序总是对少量页面多次访问 1 861 1 RW- 75

当将一个虚拟地址放入MMU中转换成物理地址时

硬件首先将该虚拟页号与TLB中的所有表项同时(并行地)进行匹配

如果存在一个有效匹配并且要进行的访问操作不违反该页的保护位

则将页框号直接从TLB中取出而不必再访问页表

如果存在匹配但指令试图在一个只读页面上进行写操作

则产生一个保护错误, 与对页表进行非法访问的表现一致

如果在TLB中不存在有效匹配项, 即对TLB的访问未命中

则进行正常的页表查询, 随后从TLB淘汰一个现有表项, 装入最新找到的页表项

于是再次访问该页面时, 将在TLB命中

而当清除TLB中一个表项时, 只需将TLB中的修改位更新到页表项, 并更新访问位

软件TLB

过去假设对TLB的管理与TLB的失效处理都完全由MMU硬件实现, 只有缺页中断会陷入内核OS

但是很多现代的RISC机器, 如SPARC, MIPS, 几乎所有的页面管理都在软件中实现

OS中TLB被显式地装载, 当发生TLB失效时, 生成一个TLB失效并将其提交给OS解决

必须在有限的几个指令完成, OS找到页面, 从TLB淘汰一个项, 装载新项, 执行出错指令

优点是, 获得一个非常简单的MMU, 为CPU芯片的高速缓存等性能改善设计腾出空间

在TLB大到可以减少失效时, TLB的软件管理就变得足够有效

一种改善采用软件TLB管理机制的机器性能的策略

在减少TLB失效的同时, 又要在发生TLB失效时减少处理开销

OS“直觉”地指出下一步可能用到的页面并预先装载到TLB表项

Operating

System - P43

处理TLB失效的常见方法是找到页表并执行索引操作以定位将要访问的~~■■■■■~~页面
但是使用软件的问题是，如果页面不在TLB中，则导致~~■■■■■~~处理过程中意外的TLB失效
注意，硬件无此问题是由于硬件使用虚拟页号并行地与TLB的所有表项进行比较
解决方案是在内存中的固定位置维护一个TLB表项的软件高速缓存

当使用软件管理TLB时，有两种不同的TLB失效
软失效 (soft miss)，指当一个页面存在于内存中，但不在TLB中的失效访问
其操作仅为更新TLB，而不需要产生磁盘I/O
典型的处理需要 10 ~ 20 个机器指令，操作时间大约花费数个纳秒 (ns)
硬失效 (hard miss)，指当一个页面不存在于内存中，即不可能存在于TLB中，而产生的访问失效
此时不仅需要更新TLB，而且需要一次磁盘存取以装入该页面
等同于虚拟内存中出现的缺页中断，需要 OS 读取一个页框，再装入待访问的页面
磁盘I/O操作处理时间远大于软失效，大约花费数个毫秒 (ms)

页表遍历

在页表结构中查找相应的映射~~■■■■■~~页表遍历
实际上会根据不同的存储结构而使用不同的搜索 (遍历) 算法
如页表索引以一个~~■■■■■~~未排序的数组存储，则可能使用线性搜索算法
以一个已排序的数组或一个二叉搜索树存储，则可能使用二分搜索算法
以一个hash表的数组存储，则可能直接通过hash值查找

实际中的~~■■■■■~~未命中的情况可能即非软失效也非硬失效

假设页表遍历并未在进程的页表中找到需要的页，从而引发一个缺页错误
次要缺页错误：指所需的页面在内存中，但并未记录在~~■■■■■~~该进程的页表里
如页面已由其他进程从磁盘调入内存，如多次调用的同一程序的正文段
此时只需在当前进程的页表中更新映射，而不用从磁盘调入

严重缺页错误：指所需的页面不在内存中，即不在进程的页表中

~~■■■■■~~类似于 TLB 管理中的硬失效，区别是进程的页表与 TLB 并不一定相同

此时需要进行磁盘 I/O，从磁盘中装入所需页面

段错误：指程序访问了一个非法地址，则无需向 TLB 中新增映射

通常由程序错误引起，OS 通常通过报告段错误来终止程序

Operating

System - P44

相比原有的内存页表方案中引入TLB以加快虚拟地址到物理地址的转换
有两种方案用于处理巨大的虚拟地址空间

多级页表

引入多级页表以避免把全部页表一直保存在内存中。

特别是对于从不需要的页表，则不应该保存在内存中。

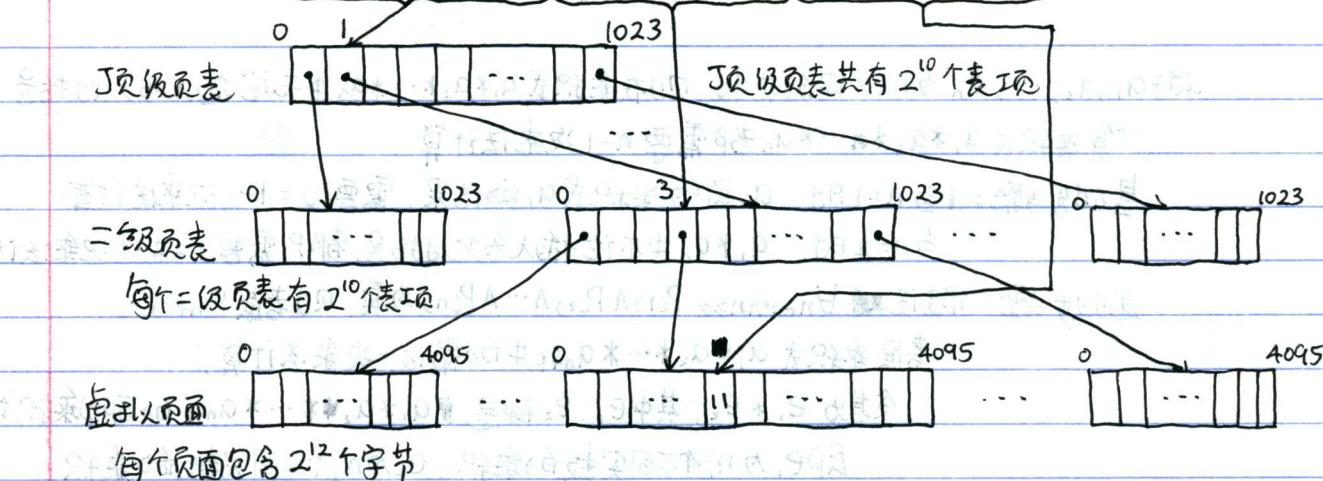
如对于一个二级页表，其虚拟地址为32位：

并拆分为10位PT1, 10位PT2, 12位偏移量3个部分

分别对应顶级页表索引, 二级页表索引, 虚拟页面索引

顶级页表(PT1) | 二级页表(PT2) | 偏移量(offset)

虚地址 0000 0000 01 00 0000 0011 0000 0000 1011 (0x0040•300B)



注意顶级页表和每个下级页表都拥有完整的页表结构。

只是除了底级页表，上层页表根据获得的并非物理地址，而是下一级页表

另外，当访问上级页表的索引的“在/不在”位为0时，会与普通内存页表一样引发缺页中断

但是此时OS并不会像普通页表一样装入页表。

而是认为程序访问了一个不希望被访问的地址。

实际上上级页表中的“在/不在”位表示该下级页表是否正在使用。

页目录

1985年的80386采用了包含页目录的二级页表机制，寻址空间达到4GB

即页目录的项指向页表，页表的项指向4KB的页框。

页目录与页表均有1024个表项，即总计寻址字节为 $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$

页目录指针表

奔腾处理器采用另一种寻址实现形式，其项指指向页目录。

但在32位下指针表仅有4项，于是总寻址字节依旧为 $2^2 \times 2^9 \times 2^9 \times 2^{12} = 2^{32}$

4级页表

64位处理器采用的寻址方式，与页目录指针表同样，页表中有512项。

总寻址空间达到 $2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$

Operating

System - P45

倒排页表 (inverted page table)，是针对页式调度层级不断增长的另一种解决方案。

相比于普通页表中每个虚拟页面对应一个页表项，倒排页表中以实际内存中的每个页框对应一个页表项。

优点是，节省了大量空间，特别是虚拟地址空间远大于物理内存时。

如对于 64 位虚拟地址，4KB 的页面，4GB 的实际 RAM。

虚拟地址空间需要普通页表有 2^{52} 个表项，而倒排页表仅需 2^{20} 个表项。

缺点是，从虚拟地址到物理地址的转换变得很困难。

当进程访问一个虚拟页面 P 时，硬件无法直接使用 P 作为页索引来查找物理页框。

而是必须搜索整个倒排页表来查找表项 (进程, P)。

而且这个搜索每个内存访问操作都要执行，而不仅仅发生在缺页中断时。

解决方案是，首先使用 TLB，用 TLB 记录频繁使用的页面，使地址转换阶段和通常转换一样快。

但发生 TLB 失效时，仍需要使用软件搜索整个倒排页表。

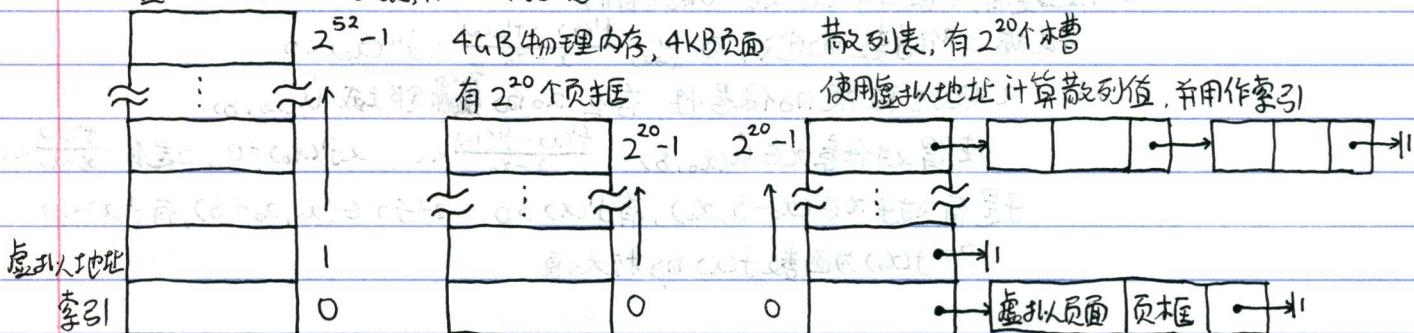
实现该搜索的一种可行方法是建立一张散列表 (Hash table)。

散列表使用虚拟地址来计算散列值 (hash value)。

将当前所有在内存中的具有相同散列值的虚拟页面链接。

如果散列中槽数与机器中的物理页面一样多，则散列表中的冲突链的平均长度应为 1 个表项的长度。

64 位虚拟地址的传统页表，有 2^{52} 个表项。



页面置换

发生缺页中断时，OS 必须在内存中选择一个页面将其换出内存，再调入访问的页面。

如果页面在内存驻留期间被修改过，则必须写回磁盘以更新磁盘上的副本。

如果没有修改过，则无需写回磁盘，而直接用调入页面覆盖即可。

可以随机地选择一个页面置换，但每次都选择不常使用的页面可以提升系统性能。

其他“页面置换”

计算机中将最近使用过的 32 字节或 64 字节存储块保存在高速缓存中，存满时需要选择一块替换。

Web 服务器将经常访问的 Web 页面放在存储器的高速缓存中，存满时需要选择一个页面替换。