分页: 介绍

分页是将进程的地址空间分割成固定大小的单元(页, page)

把物理内存看成是定长槽块的阵列(页帧, frame)

每一个虚拟内存页对应一个物理内存页帧

简单例子

分页优点1: 灵活性, 操作系统能够高效提供地址空间的抽象, 不管进程如何使用地址空间(不会假定堆和栈的增长方向)

分页优点2: 提供空闲空间管理的简单性, 操作系统会保存空闲链表, 从表中获取所需的空闲页

每个进程都有一个数据结构, 页表, 用来保存虚拟页和物理页的地址转换关系, 如示例中的 VP0--->PF3.
所以页表是每进程的数据结构, 不同进程页表中的虚拟页映射到不同的物理页(除了共享空间之外)

一个地址转换的例子, 加载地址上的数据到寄存器 %eax, movl <virtual address>, %eax

0	第0页
16	第1页
32	第2页
48	第3页
64	

0	操作系统	页帧0
16	未使用	页帧1
32	第3页	页帧2
48	第0页	页帧3
64	未使用	页帧4
80	第2页	页帧5
96	未使用	页帧6
112	第1页	页帧7
128		

虚拟地址空间 64Byte, 4 页 X 16 Byte/页

物理内存 128Byte, 8 页 X 16 Byte/页

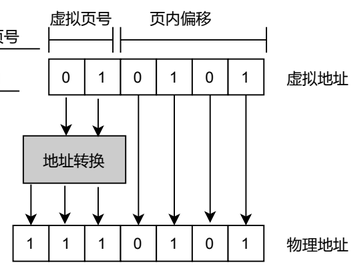
虚拟页0--->页帧3, 页1--->页帧7, 页2--->页帧5, 页3--->页帧2

1: 地址空间 64Byte, 用6位表示, 页大小16Byte 占4位, 剩下2位表示虚拟页号

2: 假设 virtual address 是 21, 二进制为 0b010101. 则 VP: 01, Offset: 0101

3: 根据页表中的地址转换关系, VP 1---> PF 111, Offset 保持不变
则转换后的物理地址为: 0b1110101

所以 movl 21, %eax 最终是从物理地址 0b1110101 取数加载到寄存器



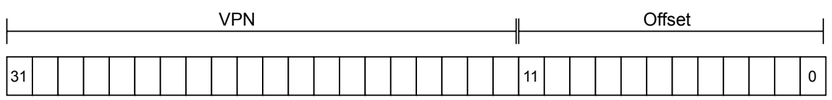
页表存在哪里

页表可以非常大, 以32位地址空间, 4KB的页为例. 虚拟地址分为20位的虚拟页号和12位的页内偏移

20位VPN, 则有2^20个地址转换, 假设每个页表条目(PTE)大小4Byte, 则页表大小 2^20X4Byte = 4MB.
如果有100个进程, 则仅页表就占用 400MB内存

页表如此之大,不可能在 MMU 硬件中存储当前运行的进程页表, 而是将每个进程的页表存储在内存中.

为操作系统保留的内存也可以虚拟化(比如32位Linux系统中1G的内核空间 0xFFFFFFFF-0xC0000000)
页表可以存储在操作系统的虚拟内存中,甚至可以交换到磁盘上



0	页表 3 7 5 2	页帧0
16	未使用	页帧1
32	第3页	页帧2
48	第0页	页帧3
64	未使用	页帧4
80	第2页	页帧5
96	未使用	页帧6
112	第1页	页帧7

页表中有什么

页表是一种数据结构,用于将虚拟地址(虚拟页号)映射到物理地址(物理帧号),最简单的形式为线性页表,
就是一个数组.VPN作为数据的Index,通过Index查找页表项(PTE),PTE中则包含PFN.

页表项内容

有效位(Valid Bit):指示特定地址转换是否有效. 例如一个程序运行时,代码和堆在地址空间的一端,栈在另一端.
中间未使用的空间则被标记为invalid.如果程序尝试访问这段空间,就会陷入操作系统,可能导致进程终止.
有效位对于稀疏地址空间的支持至关重要,通过将未使用的虚拟页标记为无效,不需要为这些页分配物理页,从而节省大量内存.

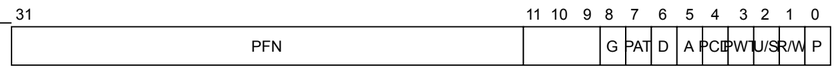
保护位(Protection Bit):指示页是否可以读取、写入或执行. 如果以不允许的方式访问页,会陷入操作系统

存在位(Present Bit):表示该页是在物理内存上还是磁盘上(即已被换出,swapped out).
该位用于研究将部分地址空间交换到磁盘上,从而支持大于物理内存的地址空间.交换允许将很少使用的页移到磁盘,从而释放物理内存.

参考位(Reference Bit):也称为访问位,Access Bit.用于追踪页是否被访问,也用于确定哪些页受欢迎,应该保留在内存中.页面替换时会非常重要

脏位(Dirty Bit):表明页面对应内存上的数据是否被修改过

一个x86架构的页表项



分页也很慢

以 movl 21, %eax 为例,系统首先从进程的页表中获取适当的页表项,执行地址转换,然后从屋里内存中加载数据

假设由硬件来执行地址转换,硬件必须知道当前运行进程的页表的起始物理地址,页表基址寄存器 PTBR.硬件将执行以下功能:

根据 PTEAddr 物理地址,硬件从内存中获取 PTE, 提取 PFN, 与虚拟地址中的 Offset结合,形成最终的物理地址,过程如下:

```
VPN = (VirtualAddr & VPN_MASK) >> SHIFT
PTEAddr = PTBR + (VPN * sizeof(PTE))
本例中, VPN_MASK=0b110000, SHIFT=4
```

因此在每一次访问内存时(指令或者数据),都需要执行一次额外的内存引用,以便从页表中获取转换的地址
额外的内存引用开销很大,可能会使进程减慢两倍或者更多

有两个必须解决的实际问题,页表占用太多内存; 页表导致系统运行变慢

```
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectionBits) == False)
    RaiseException(PROTRCTION_FAULT)
else
    Offset = VirtualAddr & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
```

内存追踪

一段C代码, 来演示使用分页时产生的所有内存访问

假设虚拟地址空间 64KB, 页面大小 1KB

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

汇编

```
1024 movl $0x0,(%edi,%eax,4) ; 0 值移动到数组位置的虚拟内存地址,地址通过取%edi 的内容并加上%eax 乘以 4 来计算
1028 incl %eax    ; %edi 保存数组的基址, %eax 保存数组索引 i, 乘以 4, 每个元素大小 4 Byte, 增加保存在%eax 中的 i
102c cmpl $0x03e8,%eax    ; 该%eax 寄存器的内容与 0x03e8(十进制数1000) 比较
1030 jne 0x1024    ;  如果两个值不相等,跳回到循环的顶部
```

需要确定代码、数组的虚拟地址, 以及页表的物理地址及内容

假设代码起始地址为虚拟地址 1024, 位于虚拟地址页的第二页 VPN=1(VPN=0 是第一页),映射到物理页帧4(VPN1->PFN4)

数组大小是 4000Byte,假设在虚拟地址 [40000, 44000), 对应虚拟页 VPN=39,40,41,42(4个1k页)

假设是线性页表,位于物理地址 1024. 页表内容只关心例子中的几个虚拟页(VPN1->PFN4, VPN39->PFN7,VPN40->PFN8,VPN41->PFN9,VPN42->PFN10)

图 18.7 展示了前 5 次循环迭代的整个过程. 自下而上分别是代码指令内存引用、数组访问、页表访问. 左边为虚拟地址,右边为物理地址

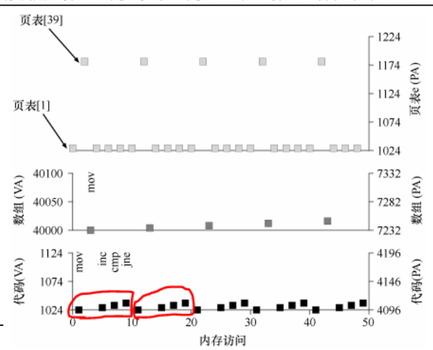


图 18.7　虚拟（和物理）内存追踪

每个指令操作将产生两次内存引用,一次访问页表查找指令所在物理地址,二次从物理地址取指令到CPU执行

mov/inc/cmp/jne 第一次访问页表[1] 物理地址1024, 取得代码虚拟地址1024 所映射的物理地址4096(页表 VPN1-PFN4), 图中最上浅色部分

mov/inc/cmp/jne 第二次次访问物理地址4096, 取指令到CPU执行.图中最下黑色部分

mov 指令有一个显示的数组赋值操作,这个会2次额外的内存访问. 先访问页表[39]取得数组虚拟地址对应的物理地址(VPN39->PFN7).
然后访问数组本身物理地址7232(PFN7). 图中中间深灰色部分

所以每个循环有10次内存访问,包括4次页表访问(获得指令物理地址),4次取指令,1次页表访问(获取数组物理地址)以及1次内存赋值

小结

相比于分段, 分页有许多优点.
首先它不会导致外部碎片,因为分页将内存划分为固定大小的页单元;
其次非常灵活,支持稀疏虚拟地址空间

分页缺点, 一是页表本身占用空间大, 造成内存浪费
二是会导致系统变慢,所有指令、数据都有额外的一次内存访问来访问页表