第9章 虚拟内存

三个重要能力

1. 将main memory看成磁盘的cache，按需在磁盘和main memory间传送数据
2. 为每个进程提供了一致的地址空间，简化了内存管理
3. 保护每个进程的地址空间不被其他进程破坏

一些概念

- PA(Physical Address)：CPU生成有效PA->传递给main memory->main memory取出data->返回给CPU
- VA(Virtual Address)：CPU生成VA->MMU转换为PA->再执行和PA类似的操作过程
- MMU(Memory Management Unit，内存管理单元)：地址翻译硬件
- VAS(virtual address space)
- PAS(physical address space)
- VP(Virtual Page)
- PP(Physical Page or page frame,页帧)
- PTE(Page Table Entry)
- $N=2^n$: 虚拟地址空间的地址数量
- $M=2^m$: 物理地址空间的地址数量
- $P=2^p$: 页的大小(bytes)
- VA = VPN + VPO
    - VPO(Virtual page offset)
    - VPN(Virtual page number)
- TLB(Translation Lookaside Buffer,翻译后备缓冲器)：是MMU中的一个小的cache
- VPN = TLBT + TLBI
    - TLBI(TLB index)
    - TLBT(TLB tag)
- PA = PPN + PPO
    - PPO(Physical page offset)
    - PPN(Physical page number)
- CO(Cache block byte offset)
- CI(Cache index)
- CT(Cache tag)
- PTBR(Page Table Base Register,页表基址寄存器)：CPU中的一个控制寄存器

VM for Cache

- VP的状态
    - Unallocated：还未分配的页，不占用任何空间
    - Cached：当前已缓存在PM中的已分配页
    - Uncached：未缓存在PM中的已分配页
- DRAM Cache 组织结构
    - VP通常很大，4KB~2MB
    - Fully associative
    - 更复杂精密的替换算法

    - Write-back instead of write-through
- Page Table
    - 常驻于PM，由PTE数组构成
    - PTE由一个valid bit和一个n位地址字段组成
    - valid bit表明该VP是否被cached
    - 如果cached，地址字段即为PPN
    - 如果uncached，null表明未分配，否则指向disk address
- Page Hit：MMU将VA作为索引定位到PTE，读取它并查看valid bit，如果cached，就使用PPN，构造出PA
- Page Fault(缺页)：
    - MMU使用VA索引定位到PTE，从valid bit判断未缓存，触发page fault exception
    - 调用内核中的page fault exception handler，选择一个victim page，如果它已被修改，就先将它复制回disk，然后修改PTE
    - 内核把目标page从disk复制到PPN，更新PTE，返回
    - 重新启动指令，把VA重新发送给MMU，正常处理
- Allocating Page：在磁盘上创建空间并更新相应的PTE，不在DRAM上
- locality保证了程序总是趋于在一个active page或者working/resident set上工作；特殊情况可能出现thrashing(抖动)

VM for Memory Management

- 简化链接：独立的地址空间允许每个进程的内存映像使用相同的基本格式，而无须关心实际存放在哪里
- 简化加载：很容易向内存中加载可执行文件和共享对象文件，只需分配VP，标记为uncached，并指向文件位置即可
- 简化共享：只需调整PT中VP映射到相同的PP即可共享
- 简化内存分配：分配连续的VP可以映射到不连续的PP

VM for Memory Protection

- 在PTE中添加额外许可位控制访问
- SUP：是否必须超级管理员才可以访问
- READ：可读
- WRITE：可写
- 违反许可条件，CPU会触发protection fault，Linux shell一般报告为"segmentation fault"

Address Translation

- page hit的过程
    1. processor产生VA给MMU
    2. MMU生成PTEA，请求PM
    3. PM返回PTE给MMU
    4. MMU构造PA给PM
    5. PM返回数据给processor
- page fault的过程 1-3. 同上 4. PTE中valid bit为0，触发exception，控制权传给内核handler 5. handler确定victim page，如果已修改则换出到disk 6. handler调入新page，更新PTE 7. 返回到原来的进程，CPU重新发送VA
- 结合cache和VM：即在MMU和PM之间增加了一层cahe
- 利用TLB加速
    1. CPU产生VA

    2. MMU从TLB取出PTE
    3. MMU将VA翻译成PA给PM
    4. PM将数据返回CPU 如果TLB miss，则MMU必须从L1 cache中取出相应的PTE，放在TLB中
- 多级页表
    - VA被划分为k个VPN和1个VPO
    - 节约巨大空间
    - 实际并不比单级页表慢很多

Intel Core i7/Linux Memory System

- Core i7 Address Translation
    - CR3控制寄存器指向L1 PT的起始位置
    - PTE中P表明是否在PM中缓存
    - R/W可读可写
    - U/S用户超级
    - XD禁止执行
    - A(reference bit)实现时钟策略的页替换算法
    - D(dirty bit)是否需要写回
- Linux Virtual Memory Area
    - 每个进程会有一个task_struct
    - task_struct中有一项mm，指向mm_struct,描述了虚拟内存的当前状态。
    - mm_struct中pgd(page global directory)指向第一级页表基址
    - mm_struct中mmap指向一个vm_area_structs的链表，描述了当前虚拟地址空间的情况
    - vm_area_structs中vm_start,vm_end指向区域的起止处，vm_prot描述读写权限，vm_flags描述共享私有，vm_next指向下一个区域结构
    - 缺页异常处理
        1. VA合法吗？如果不在vm_start，vm_end定义的区域内，触发Segmentation fault
        2. 内存访问合法吗？只读页面中写，用户模式读内核，会触发Protection exception
        3. 合法页面。选择victim page并重新发送指令

Memory Mapping

- 映射类型
    - Regular file: demand paging，没有实际交换进入物理内存，直到CPU第一次引用到页面；如果有空余用0填充
    - Anonymous file：demand-zero page，由内核创建，驻留在内存中
    - 共享对象：copy-on-write
    - mmap：

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

//成功返回映射区域指针，出错返回MAP_FAILED(-1)
//内核最好从地址start开始，将fd指定的对象从offset偏移开始连续length字节，映射到虚拟内存
//start不一定被满足，通常定义为NULL
//prot描述了访问权限，包括PROT_EXEC,PROT_READ,PROT_WRITE，PROT_NONE(不可访问)
```

```
//flags:MAP_ANON-匿名demand-zero，MAP_PRIVATE-私有copy-on-write，
MAP_SHAERED-共享对象
void *mmap(void *start, size_t length, int prot, int flags, int fd,
off_t offset)

//删除从start开始的length字节组成的区域
int munmap(void *start, size_t length);
```