操作系统第九章笔记：Virtual Memory

Zhitao Yin

2024/11/24

1 Background

1.1 Page fault 缺页

在第八章的 pagetable 中已经介绍过真实 OS 中的 pgtbl 包括 valid bit ,modified bit(dirty bit), control bit。这里详细拓展 control bit:control bit 中有一位是 use bit 表示上一次被访问后经过的时间，use bit(有的教科书也叫 reference bit) 是用来决定 Roll in 和 Roll out 的字段。

DEFINE Page fault 缺页

操作系统中有虚拟地址空间和物理地址空间，LA 就属于虚拟地址空间。虚拟地址空间没有实体，LA 本身也没有意义。当 OS 需要调用某一页时，如果根据 LA 得不到相应的 PA 或者得到的 PA 并不是我需要寻找的帧，那么就会出现 load page fault 缺页 error。

1.2 缺页中断机制

当缺页 error 出现时，系统提出缺页中断，然后由 OS 的缺页中断处理程序处理中断。

首先将缺页的进程阻塞，挂到阻塞队列中，直到 paging Schedule 后再调回该进程，让其进入就绪队列。

如果内存中有空闲块，则为该进程分配此空闲块，并将缺页的页面 Load 到这一块中，然后修改页表，对应映射。

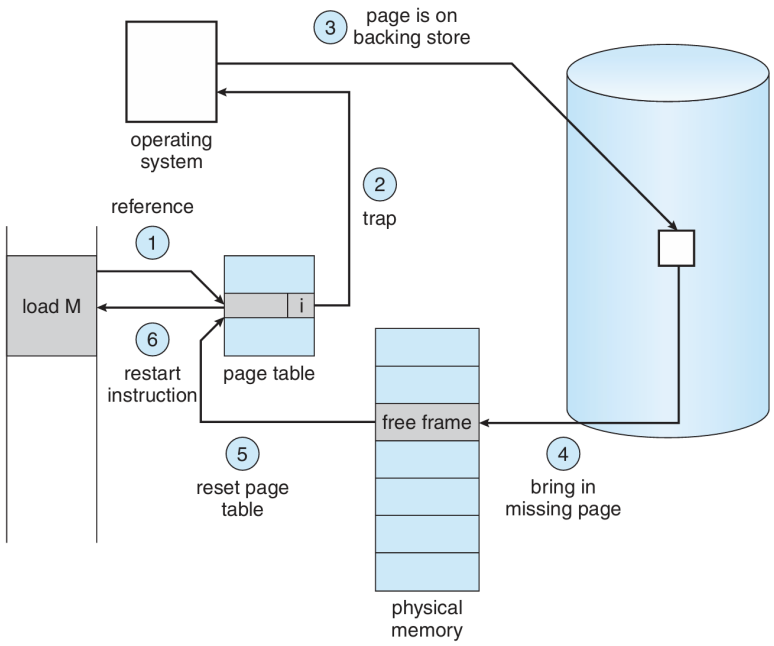

图 1: 缺页处理机制

2 Demanding paging

page fault 缺页出现时，OS 会按照如下步骤处理请求页。

页号	内存块号	状态位	访问字段	修改位	外存地址
0	无	0	0	0	x
1	b	1	10	0	y
2	c	1	6	1	z

图 2: 页表项结构

> 解释

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

2. If the reference was invalid, we terminate the process. If it was valid but wehave not yet brought in that page, we now page it in.

3. We find a free frame(bytakingonefromthefree-framelist,for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process andthepagetabletoindicatethatthepageisnowinmemory.

6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory

2.1 Demanding Paging Performance 请求页的效率

Assumption：缺页率 p(Page fault rate，就是访问页时发生 page fault 的平均概率)，页面访问时间 ma, 处理 page fault 时间为 t

effective access time $= (1-p) \cdot ma + p \cdot t$ 平均访问时间 page fault time = page fault overhead+ swap out 1 page + swap in 1 page + restart overhead

overhead: 开销

3 Placement Policy

当物理帧几乎被占满时，这时又有新的页需要分配帧。换下哪些页？

page fault time 中缺页率很重要，如何削减缺页率？介绍几种算法

3.1 opt policy 最佳置换算法

原理：每次被换掉的页是**今后不被使用或者最少被使用的页**。

3.2 FIFO

原理：每次淘汰最先进内存的页。

3.3 LRU:least recently used 最近最少被使用

原理：淘汰最近最久未经使用的实现方法：用一个变量 tmp 记录上次使用过后经过的时间。每经过一个时间片 Tmp+1, 每当被使用 tmp 置 0，

每当被移入 tmp 置 0，每当被移出 tmp 置零。

3.4 CLOCK

CLOCK 算法又称 NRU 算法，最近未用算法 not recently used。

简单位 CLOCK 算法实现方法：

为每个页面设置一个访问位，再将内存中的页面通过链接指针链接成一个循环队列。当某页面被访问时，其 use bit 为 1。当需要淘汰一个页面时，只需检查页的访问位。如果是 0，就选择该页换出；如果是 1，则将它置为 0，暂不换出，继续检查下一个页面，若第一轮扫描中所有页面都是 1，则将这些页面的访问位依次置为 0 后，再进行第二轮扫描（第二轮扫描中一定会有访问位为 0 的页面），因此简单的 CLOCK 算法选择一个淘汰页面最多会经过两轮扫描。

NRU 算法实现：使用 use bit 和 modified bit

1. (1,0) 最近被访问过，没修改过，安全。
2. (0,1) 最近没被访问过，但是刚被修改过，也可能很久以前被修改过，但有马上被使用的风险，比较危险。
3. (0,0) 没被访问，也没修改过，极度危险。
4. (1,1) 被访问了，也被修改了，安全。

NRU 算法步骤如下：

1. 从当前位置开始扫描到第一个（0,0）的帧用于替换。本轮扫描不修改任何标志位。
2. 若第一轮扫描失败，则重新扫描，查找第一个（0,1）的帧用于替换。本轮将所有扫描过的帧访问位设为 0。
3. 若第二轮扫描失败，则重新扫描，查找第一个（0,0）的帧用于替换。本轮扫描不修改任何标志位。
4. 若第三轮扫描失败，则重新扫描，查找第一个（0,1）的帧用于替换

由于第二轮已将所有帧的访问位设为 0，因此经过第三轮、第四轮扫描一定会有一个帧被选中，因此改进型 CLOCK 置换算法选择一个淘汰页面最多会进行四轮扫描。

可能你会觉得有点枯燥，上网找个视频看看吧，很简单的算法。

3.5 LFU 和 MFU

least frequently used | most frequently used

3.5.1 LFU

LFU 可能比较好理解，访问次数最低的页被移出去。

实现方法就是变量 count，每次被访问就 count++，每次移出去 count 最小的页。

坏处是：通常被移出去的都是刚进来的页，因为还没被访问用过。

3.5.2 MFU

MFU 的理念就是，你被用的次数这么多了，说明你可能已经 Over used 了。应该就快被用干了，应该不会再被使用了。

3.6 Page buffering 页面缓存

被置换的页不会马上被移出主存，而是暂放在两个 list 中 free page list 和 modified page list。分别放被置换的，没有修改过和修改过的 page。

每当出现 page fault 的时候，首先检查需要的 page 在不在这两个 list 里面，如果在，那么直接取出来。如果不在，首先找到这个页，然后把 free page list 的 head 存放的帧位置给这个 needed page, 然后更新 free list，把 head = head->next。然后暂时保留原本要 replace 的那一帧，把他放入 list 尾 (根据有没有修改)

3.6.1 cleaning policy

所以什么时候释放 modified page 呢？

1. Demand cleaning

   被置换的时候写回外存 disk

2. precleaning

   大批量的帧直接提前写回

但是考虑到 page buffering 页面缓存，那么采取的策略是：当这帧即将被替换时，先写回 disk，然后作为 freepage 写到 freelist 尾部。

4 Frame allocation policy 页面分配策略

4.1 Background

OS 必须知道要给一个进程分配多少帧才算合适。分的帧多了，那么 OS 的空间利用率太低；分的帧少了，缺页率太高。所以引入下方知识。

4.2 allocation and Scope

1. fixed allocation policy

   固定分配，每个进程分配一样多的帧

2. variable allocation policy

   可变分配，动态分配帧数

3. local replacement policy

   局部置换：进程 A 只允许使用那些分配给进程 A 的帧，所以缺页？自己拿一页出来，空闲一帧然后放进去

4. global replacement policy

   全局置换，只要主存里面有空闲帧就给你进程用。

由含义你可以看到：固定分配和全局置换是不能同时使用的。你一个进程只能用 x 帧，不给你用多余空闲帧。而且大多数 OS 都用的可变分配 + 全局置换。但是这有个问题。

如果你用可变 + 全局，那么当空闲帧用完之后，OS 会随机分配一个别的已经被占领的帧给你用。所以这种策略 Page fault rate 很高。

Windows 用的是可变 + 本地。这是最好的策略。

4.3 Thrash 抖动现象

如果采取本地置换，当进程频繁使用的页被经常调出，因为帧数不够时，缺页率极高，这种情况叫 thrash。所以问题还是，怎么动态分配合适的帧？

4.4 Working Set Strategy 和 page fault frequency Strategy

4.4.1 Work Set

Work Set 工作集：在规定的某段时间内 Δt，进程实际使用的页面的集合。举个例子,ABCCBADBDBABCDEF,Δt 时间内用了这些页，那么 Work Set 集合就包含 6 个元素，ABCDEF。

工作集策略就是，当我规定好时间区间后，我把 proc 过一遍，得到最大的 Work Set，你分配的帧数必须大于等于 Working set 的元素个数。

很合理，对吧。

4.4.2 page fault frequency Strategy

原理是：对进程从 0 到无穷分配帧数，你设定一个区间 [L,U]，然后如果 page fault rate 在这个区间内，你就取对应的帧数。

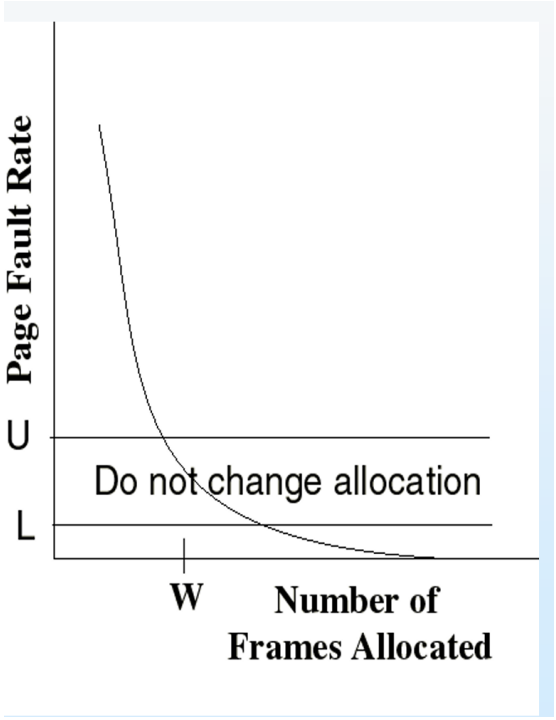

图 3: 帧数-缺页率图

4.4.3 load control

首先介绍 resident set.

resident set 驻留集：指当前进程驻留在物理内存中的，被 OS 分配的内存块的集合。

为了保持 OS 的 multiprogramming 的能力，每次可以装入多少进程也应该是受控制的。也就是 Load control。首先我们通过 page fault frequency Strategy 可以知道，帧数分配的越多那么效率就越高。而 workset 告诉我们，一个大进程会索要很多的帧以减少缺页率。

5 Copy on Write

Copy-on-write 的核心思想就是延迟复制：只有在数据实际被修改时才进行复制。换句话说，当多个进程或线程共享相同的数据时，系统会首先共享这些数据，直到某个进程或线程需要修改数据。

5.1 解释

当进程 fork() 之后会产生一个子进程，论上，子进程会是父进程的一个完整拷贝，包括其内存空间、打开的文件描述符等。但事实上，复制整个进程的内存会消耗大量资源，因此现代操作系统使用了 COW 策略。当父进程调用 fork() 时，操作系统并不会立即复制父进程的内存，而是让父子进程共享相同的内存区域。这意味着，直到其中一个进程试图修改内存时，操作系统才会为该进程创建内存的副本（即“写时复制”）。这种机制大大减少了 fork() 调用时的开销。

Copy-on-write 还被广泛应用于共享内存和内存映射文件的实现中。例如，在将文件映射到内存时，操作系统可能会将文件内容映射到多个进程的虚拟地址空间。这些进程可以共享同一段内存区域，当某个进程修改这些内容时，操作系统会将该进程修改的数据拷贝到新的内存位置，而其他进程则不会受到影响。

6 Mapped files 映射文件

6.1 Tradition file I|O

我们知道，文件 files 通常存在 disk 硬盘中，而 disk 作为外存，其与 cpu 信息交换的速度特别慢，也就是说 Read or Write 很慢。因此我们引入 mapped file 方式，将 file 文件映射到主存中 (你可以理解为把 file 中的 data copy 到 main memory 中)

6.2 Memory Mapped files

映射的内容很简单，无非是把 disk file copy 到主存中。但是你能否回答以下问题？

1. 什么时候进行映射
2. 映射之后，proc 如何访问这个 file 的 copyed data
3. 映射之后，这个 file 以什么形式存在？是存在于虚拟地址空间？还是 main memory 中已经给 file data 分配了一定内存？

下面我将详细解释 memory mapped file 的过程。

首先，当一个进程需要访问 disk 中文件时，在进程即将被执行时，也就是 after compiled,before executed，cpu 读取到 read/write 指令, 这时 cpu 得到访问 file 的请求，cpu 将给 file 一个 LA。

我解释的不是非常清晰，cpu 给定 file 一个 LA，但是 file 的数据放置在哪？我们现在已经分配了 LA，但是 dat a 其实还并没有载入物理空间，因此，当且仅当 proc 被真正执行的时候，LA 变成 PA 时，才会 copy filedata。

由于 file 的共享性，可能很多个 proc 都需要 read 或 write file。只要让 proc-s 中 file 对应的 LA 都是相同的即可 (这种情况要么是 copy on write，要么是 file 只读)。不允许多个 proc 同时 write file，需要 write file 的话，参考 cow，再用一个页复制 data。

当进程几乎结束时，也即是 file write 已经完成时，需要将 file 写回到 disk 中。这种情况参考 page buffering 页面缓存。