Lab 实验报告 VI

Lab: Copy On Write

丁睿

dromniscience@gmail.com

更新：2020 年 12 月 1 日

目录

1 任务完成清单

Subtask	Done?	Time
Implement copy-on write	Y	3h + 6h

Grade: 110/110

其中 **Time** 列的 3h 是指构思和设计耗费的时间，余下 6h 则是实现与调试的时间。

我的作业已开放在这个网页上。

2 任务特点

本次 lab 的任务描述非常宏观，也没有划分成多个子任务，这导致如下两个特点:

- 可行的设计方案很多。
- 事先的规划相当重要。

另一方面，为使 COW 与已有的存储管理和 trap 处理兼容，必须仔细地搜索并决定源码修改的地点。除此以外，就如同任何的系统程序那样，必须注意到以下三点内容:

1. 代码复用
   例如本次的 page fault handler, 它将在 `usertrap` 和 `copyout` (kernel/vm.c) 中为处理对 COW 页的写。因此应当被包装成一个鲁棒的处理函数，就如同 `syscall` 和 `devint` 那样。
2. 安全检查 & 错误处理
   使用`PTE_COW`就是一个很好的例子，尽管通过每个物理页的引用计数就也可以实现 COW 的检测[1]。除此以外，user/usertests.c 中的 copyout 专门测试了对一个数值相当大的非法地址和 guard page 的访问，execout 专门测试了没有空余物理页的情况。所有这些必须经过详细的检查，以及设计错误处理机制。
3. 并发访问
   由于支持多核的并发环境，对于`refcnt`的访问和修改必须保证原子性。例如，正如我们稍后就将看到的那样，一个对引用计数自增或者自减的操作必须同时返回修改前 (或者修改后) 的值，否则就可能产生多次加入空闲页链表的错误。
   尽管 `make grade` 没有显式地测试这种竞争，但是我们必须注意到这样的问题。

以下我将首先阐述我的设计方案，尤其是隐藏在最终实现之外的折中考虑。接下来将围绕以上提及的三点注意说明当前设计中容易出错的细节 (有一些也是在我 debug 的过程中找到的)。尽管别的设计方案下某些结构不尽相同，但是我挑选出来的这些细节我认为几乎所有可行的解决方案都无法避免。

[1]这个断言是有问题的。假设一个父进程在 `fork` 出子进程后写一个 COW 页前其子进程就结束了，那么实际上该页仍然是 COW 页，但是引用计数为 1!

3 总体设计

唯一能够发生 COW 的场景在于 `fork` 出的子进程与父进程共享相同的页表映射 (子进程有独立的页表，但此时与父进程指向完全相同的物理页)。由于 `uvmcopy` 被专门地使用做子进程页表和物理页内容的初始化，我们只需修改它而不必修改 `fork` 就能利用写时复制的技术。

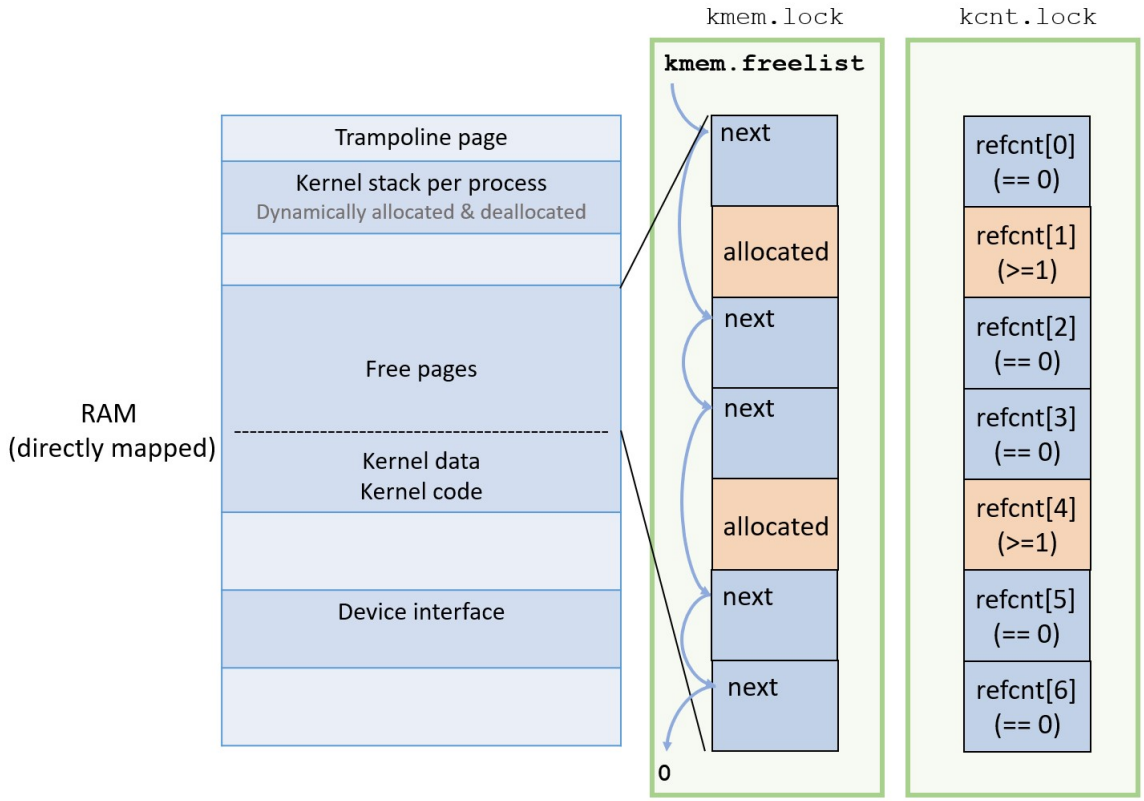

图 1: 为每个物理 RAM 页匹配一个引用计数。绿框表示用自旋锁保证互斥地访问。kmem 列里的每个框表示一个物理页。橘色表明已被 `kalloc` 分配，蓝色表示该页未分配。空闲页被组织成链表，其中的后向指针占用空闲页本身的空间。kcnt 的每个框表示计数相应物理页的数组元素 (它们实际存储在 RAM 里的 kernel data 区域)，其中空闲页对应的引用计数为 0，已分配页的引用计数至少为 1。

由于一个也可以被多个进程引用，有必要维护一个引用数组。实际我们需要进行引用技术的区域如上图所示。多核并行的访问冲突使我们必须采取锁加以保护。当然，可以为每个数组元素维护一把锁，但考虑到 RAM 的容量，这显然带来极大的浪费。况且自旋的情况并没有那样频繁，因此统一使用一把锁进行保护是令人满意的。

一切内存页的释放都必须通过 `kfree` 函数，而一切内存页的分配都必须通过 `kalloc` 函数。前者使我们只需要在 `free` 中自减引用计数，到零时真正地释放该页，而不需要修改其他企图释放页的代码。后者是需要自增引用的一种场合，而且自增后的引用值必然为 1。引用自增的另一种情形则是 COW。

最后，如果一个页试图直接通过 `write` 系统调用向 COW 页写，那么我们还要在 `copyout`

函数中分配新的 COW 页。

为了代码的简洁，我们设计缺页处理程序遇到 COW 页时总是分配新的页，然后从 COW 页拷贝内容，即使该 COW 的引用计数已经是 1。然后它修改进程的页表项的物理地址和权限位反映这一变化，然后调用 `kfree` 释放 COW 页[2]。

4 细节分析

从核心的缺页处理函数看起，它将引出其他函数设计的某些约束。我将它声明为`int pf_handler(uint64 va, int check);`其中 `va` 是一个不需要对齐的用户虚拟地址，`check` 表示它是否检查报错页的`PTE_W`和`PTE_COW`位。`usertrap`当然应该以`check`有效调用它。而`copyout`应以`check`无效调用它，使得我们调用前根本不用保证发现了一个向 COW 的写。在此之前，我们总应该检查用户地址的合法性，例如过高或者向用户栈的 guarding page 中写。一旦出错，它就返回-1，否则返回 0。这些统一的设计让我们的代码做到了简洁而安全。

一旦`copyout`检测到了负的返回值，它也立刻返回-1，以使`syscall`发现它并返回负值 (就像正常出错的系统调用那样)。而`usertrap`检测到这个事情，就应当标记`p->kill`以杀死进程，无论是因为地址不合法还是没有空余的内存页。

最后，我们提到 RISCV 支持三种缺页错误 ([2.] Page 62):

Type	**Exception code** (in `scause`)
Load page fault	0xD
Store/AMO page fault	0xF
Instruction page fault	0xC

由于第一种和第三种绝不应触发 COW 缺页，因此`usertrap`只需判断如下条件:

```
if (r_scause() == 0xF){
    // COW
}
```

接下来我们将要提到如何维护`kcnt`。

```
/* kernel/kalloc.c */
struct {
    struct spinlock lock;
    int refcnt[(PHYSTOP - KERNBASE) / PGSIZE];
} kcnt;
```

[2]这意味着 COW 页的标志位永不改变。它是合理的。如果三个进程共用一个 COW 页，那么缺页处理程序必须根据 COW 页的引用计数决定是否恢复 COW 页的写权限并取消`PTE_COW`标志。这就使代码更容易出错。

为了减少数组大小，我采用了偏移加移位的方法。即首先将物理地址减去 RAM 起始偏移，再除以页大小，如下:

```
/* kernel/riscv.h */
// Physical address to INDEX
#define PA2IDX(pa)      ((((uint64) (pa) - KERNBASE)) >> 12)
```

任何对于该数据的访问都要由自旋锁保证互斥。它们支持取值 (get_rc)、自增 (incre_rc) 和自减 (decre_rc) 操作。其中自增自减应当同时实现运算和值返回，否则将在kfree中引入竞争。

为了说明理由，设想如下场景。这时kfree应当先自减页面引用，然后再访问其值，如为零则将其加入空闲页的链表中。不幸的是，父子进程可能在两个核上并行地执行对同一个 COW 页的kfree。它们首先依次对引用减 1，然后就释放锁。接着又依次取得锁来访问引用计数。但这时它们都会得到 0，因此该页将被加入链表两次！即使kfree被设计成先访问页面引用，再减 1，然后根据刚才的值是否为 1 来选择将页加入空闲页链表，还是能找到这个页最终未被释放的竞争可能。**只要将值修改和值访问打包成一个原子操作，竞争就不复存在。**

如下函数的返回值声明说明了这一点:

```
/* kernel/kalloc.c */
int get_rc(void *pa);
int incre_rc(void *pa);
int decre_rc(void *pa);
```

最后我们简单地提及引用计数的初始化。注意freerange调用kfree初始化链表，因此这些页面引用应当先被设成 1。这样我们能够方便地复用kfree的代码，并把代码量安全地压缩到了几乎最少。

我们的实现中还加入了不少判断以方便调试。例如，kalloc总是要求自己分配的页的计数为 0、decre_rc要求操作后的数值非负。再如缺页处理程序要求当前虚址存在于用户空间等等。在实际调试时，它们帮了我大忙！

5 参考资料

1. *xv6: a simple, Unix-like teaching operating system*
   Russ Cox, Frans Kaashoek, Robert Morris *August 31, 2020*
2. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*
   Document Version 20190608-Priv-MSU-Ratified
3. *The RISC-V Reader: An Open Architecture Atlas*
   David Patterson, Andrew Waterman *1st Edition*
4. 现代操作系统 [M]
   A.S.Tanenbaum, H. Bos 著, 陈向群 马洪兵 译, 北京: 机械工业出版社, 2011: 47-95