# Assignment 3 - CoW: Copy On Write

2025 年 4 月 16 日

## 1 Background

Virtual memory provides a level of indirection: the kernel can intercept memory references by marking PTEs invalid or read-only, leading to page faults, and can change what addresses mean by modifying PTEs. There is a saying in computer systems that any systems problem can be solved with a level of indirection.

This assignment explores an example: copy-on-write fork.

**Quote** "All problems in computer science can be solved by another level of indirection."

计算机科学中的所有问题都可以通过增加一层间接层来解决。

## 2 The problem

The `fork()` system call in xv6 copies all of the parent process's user-space memory into the child. If the parent is large, copying can take a long time. Worse, the work is often largely wasted: `fork()` is commonly followed by `exec()` in the child, which discards the copied memory, usually without using most of it. On the other hand, if both parent and child use a copied page, and one or both writes it, the copy is truly needed.

## 3 The solution

Your goal in implementing copy-on-write (COW) `fork()` is to defer allocating and copying physical memory pages until the copies are actually needed, if ever. COW fork() creates just a pagetable for the child, with PTEs for user memory pointing to the parent's physical pages. COW fork() marks all the user PTEs in both parent and child as read-only. When either process tries to write one of these COW pages, the CPU will force a page fault. The kernel page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to

Build commit hash: 4046c24

refer to the new page, this time with the PTE marked writeable. When the page fault handler returns, the user process will be able to write its copy of the page.

COW fork() makes freeing of the physical pages that implement user memory a little trickier. A given physical page may be referred to by multiple processes' page tables, and should be freed only when the last reference disappears. This mechanism is called **reference counting** (引用计数).

## 3.1 Overview Guidance

1 Modify `mm_copy()` to map the parent's physical pages into the child (see `mm_mappages_cow()`), instead of allocating new pages. Clear PTE_W in the PTEs of both child and parent for pages that have PTE_W set.

2 Modify `handle_pgfault()` in *os/trap.c* to recognize page faults from usermode. When a write page-fault occurs on a COW page that was originally writeable, allocate a new page with `kallocpage()`, copy the old page to the new page, and install the new page in the PTE with PTE_W set.

3 Ensure that each physical page is freed when the last PTE reference to it goes away – but not before.

4 A good way to do this is to keep, for each physical page, a "reference count" of the number of user page tables that refer to that page. Set a page's reference count to one when `mm_mappages()` allocates it from `kallocpage()`. Increment a page's reference count when fork (`mm_copy`, `mm_mappages_cow`) causes a child to share the page, and decrement a page's count each time any process drops the page (`freevma()`) from its page table. `kfreepage()` should only be called if its reference count is zero, i.e., no more user processes are using it.

5 It may be useful to have a way to record, for each PTE, whether it is a COW mapping. You can use the `RSW` (reserved for software) bits in the RISC-V PTE for this.

6 If a COW page fault occurs and there's no free memory, the process should be killed.

7 Pages that were originally read-only (not mapped PTE_W, like pages in the text segment) should remain read-only and shared between parent and child; a process that tries to write such a page should be killed.

8 Modify `copy_to_user` to use the same scheme as page faults when it encounters a COW page.

**Reference Counting** 引用计数（Reference Counting）是一种共享资源管理技术，用于追踪和管理共享对象的**生命周期**。其核心思想是，每当一个对象被引用时，它的引用计数就增加；每当一个引用不再使用该对象时，引用计数就减少。当一个对象的引用计数变为零时，说明没有任何地方再引用该对象，此时系统可以安全地释放这个对象占用的资源。

### 3.2 Implementation Hints

1 你可以在代码仓库中搜索关键字 `Assignment 3`，观察代码中的提示。

2 我们仅对用户页面（不包含页表所用的页面）进行引用计数。引用计数相关的代码框架已在 *vm.c* 中给出。

如果你查阅到相关资料，其表示可以在 *kalloc.c* 或 *vm.c* 中实现引用计数。但是，在该作业中，**请勿在 kalloc 模块中实现引用计数**，防止我们的 ktest 模块失效。

3 xv6 是如何进行用户页面的回收：

- `freeproc` 调用 `mm_free`。
- `mm_free` 会调用 `mm_free_vmas`，对于 vma 链表中的每个 vma，调用 `freevma` 进行释放。
- `freevma` 会找到它所表示的虚拟内存区域对应的物理页面，并进行释放。
- 然后，`mm_free` 调用 `freepgt`，释放页表自身所用的物理页面。
- 最后，`mm_free` 会释放 `struct mm` 结构体。

4 xv6 会在 `mm_mappages` 和 `mm_remap` 两处函数中为用户进程分配物理页面。

5 在本次作业中，我们简化了 `mm_remap` 函数，使其永远只能用于扩充 VMA。用户程序会使用 `sbrk` 系统调用调整 brk（即 Heap 区域）的大小，该系统调用会修改 `p->vma_brk` 所表示的区域，并使用 `mm_remap` 重新映射该 VMA。

6 你可能会在 C/C++ 课了解到，在使用时 `malloc` 和 `free` 动态分配对象时，小心两种情况：Use-after-free 和 Double Free。前者表示你在释放一个对象后，仍然使用了它；后者表示你对同一个对象进行了两次释放，它会损坏内存分配器。

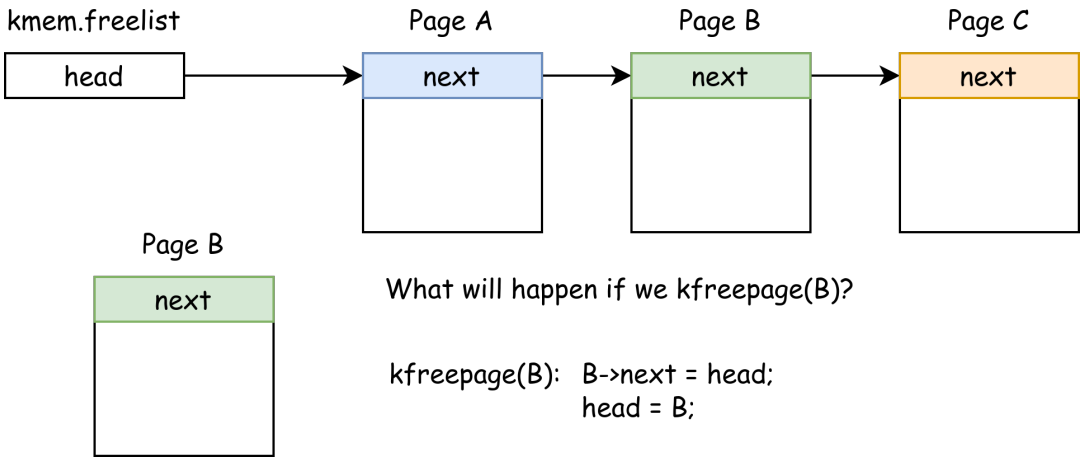
7 在内核中，我们使用 **kalloc** 模块来动态分配物理页面。用户页面即上述进行“动态分配”的对象。

8 当你忘记引用计数时，你大概率会遇到 Use-after-free、Double Free 和内存泄露，前两种情况通常会导致内核进入非常奇怪的状态。考虑以下情况：

- 在共享一份物理页面时没有增加引用计数。某个物理页面被两个进程共享，而引用计数仅为 1。在其中一个进程退出时，引用计数减为 0，物理页面被释放。此时，另一个进程仍然在使用该物理页面。
- 在释放页面时没有考虑引用计数。导致一个共享的页面被释放两次，这会破坏 **kalloc** 模块。

9 `kallocpage/kfreepage` 详解：我们使用单向链表来链接所有空闲的物理页面，在每个页面的开头放置一个 `struct linklist` 结构体，并在内核保存链表头指针 `kmem.freelist`。由于只有空闲页面才会有 `next` 指针，对于被分配的页面，用户不会也不应该关心页面头部曾今有个 `next` 指针。

所以，思考 Double-free 的情况下会发生什么：A,B,C 是三张已分配的页面，考虑如下 `kfreepage` 顺序：C, B, A, B，最后一次 `kfreepage(B)` 是一次 Double Free。




然后，考虑 Double-free 后，我们再次调用三次 `kallocpage` 会发生什么？我们鼓励你在报告中回答这个问题。

### 3.3 Debugging Hints

1 跟踪 `kallocpage/kfreepage`：你可以在 *kalloc.c* 的 **第一行**（在所有 `#include` 之前）加入 `#define LOG_LEVEL_DEBUG`，内核会在调用这两个函数时打印相关信息。

2 在上述日志中，我们可以找到函数的调用者：*alloc/free PA, by caller*。*caller* 是一个 `0xffffffff8020xxxx` 的地址，这是调用这两个函数时 ra 寄存器的值，该地址为内核的代码地址。你可以在 *build/kernel.asm* 中找到它对应的汇编，你也可以用 `addr2line` 工具快速定位它对应的 C 代码行号。

```
[DEBUG 0,2] kfreepage: free: 0x0000000080d14000, called by 0xffffffff80208b20
[DEBUG 0,2] kfreepage: free: 0x0000000080d13000, called by 0xffffffff8020954c
sh >> QEMU: Terminated

/D/s/a/src $ addr2line -e build/kernel
0xffffffff8020954c      <-- Paste and Enter.
~/a3-cow/src/os/vm.c:228
0xffffffff80208b20
~/a3-cow/src/os/vm.c:209 (discriminator 1)

```

3 Use-after-free 和 Double-free 实在是令人头疼，它们会导致内存错误，并且在错误的第一现场中隐身，但是导致其他进程发生非预期的行为（第二现场），那么怎么 Debug 呢？

**追踪 Memory Corruption**：在从 **kalloc** 模块分配页面时，我们会将返回的新页面填充上特定的垃圾字节 (0xaf)；在释放页面时，我们也会将页面填充上特定的垃圾字节 (0xdd)。

这样，我们可以在第二现场（通常是 Kernel Panic）时注意到内存或寄存器中全是垃圾字节（这通常发生在 PageFault，因为页面上存储的指针被垃圾字节覆盖了），提示了我们可能遭遇了 Use-before-initialization、Use-after-free、Double-free。

**推荐阅读**：在 Linux + glibc 中，我们可以用怎样的工具来追踪这种内存错误，这些工具的实现原理是什么？在 Linux Kernel 中有类似工具吗？你被鼓励在报告中回答这个问题。

4 你也可以修改 *kalloc.c* 中的 `kfreepage` 函数，在每次释放页面时，验证该页面没有已经被释放过。这是一种防御性编程。

5 你可以使用 `vm_print` 函数打印一张页表，该函数实现在 *debug.c* 中，你需要引用头文件 *debug.h*。在用户模式，你可以使用 `ktest(KTEST_PRINT_USERPGT, 0, 0);` 委托 ktest 打印用户页表。

### 3.4 Startup

在你真正开始修改代码之前，我们建议你思考清楚以下问题，并且鼓励你在报告中回答这些问题：

1 在 xv6 中的内核内存管理模块 (*kalloc.c*) 中，哪个函数负责分配物理页面，哪个函数负责释放物理页面。

2 在 xv6 中的用户内存管理模块 (*vm.c*) 中，哪些函数负责建立用户页面，哪些函数负责释放用户页面，哪些函数负责操作用户页表。

3 在 `mm_copy` 中，我们使用 `mm_mappages` 映射子进程的页面，并从父进程复制页面内容。我们注意到这种复制可能是多余的，因为 fork 后调用 exec 会丢弃子进程在 fork 时拷贝的页面。所以，我们是如何避免这种复制的？

从 `mm_copy` 中开始，将 `mm_mappages` 改为 `mm_mappages_cow`，并在 `mm_mappages_cow` 中使用 CoW 的方式映射父进程的物理页面到子进程中。

## 4 Checkpoints

**本次作业一共 4 分，有 4 个 Checkpoint 和一份报告**。每个 Checkpoint 各 1 分，报告不占分，但是强制要求写。你的报告应该符合以下要求：

- PDF 文件格式的报告。
- 每个 checkpoint 运行成功的截图。
- 对于指引中提出的思考问题，你可以不在报告中回答，但是我们鼓励你思考这些问题并做出回答。
- 记录完成这个作业一共花了多少时间。
- （可选）描述你在完成这个作业时遇到的困难，或者描述你认为本次作业还需要哪些指引。
- （可选）你可以在报告中分享你完成这个作业的思路。
- 越简洁越好。

**要求** 你应该使用 `make run` 启动单核心的 xv6。本次作业我们提供了 `usertest`，在启动内核后，你应该能在 `applist:` 提示符后看到一项 `cowtest`。你需要通过 `cowtest 1-4`，能稳定通过即视为通过 Checkpoint。

注：用于回归测试的 `proctest` 会报错，这是预期之内的，本次作业的评分与之无关。

**Checkpoint 说明** Checkpoint 1 会在系统没有足够物理内存的情况下进行 `fork`，而 CoW 通过共享父进程的物理页面来避免分配新的物理页面。Checkpoint 2 会检查物理页面是否会泄露。Checkpoint 3 会向共享的页面写入，触发 `Copy-on-Write`，并检查子进程和父进程是否具有独立的物理页面。Checkpoint 4 会检查 `copy_to_user` 函数是否会触发 `Copy-on-Write`。

# 5 提交

将你的报告重命名为 report.pdf，放到与该 PDF 文件的同目录下，运行 `make handin`，这会生成一个 handin.zip 压缩包。上传该文件到 Blackboard 即可.