OSLAB Report

191240030 刘闵

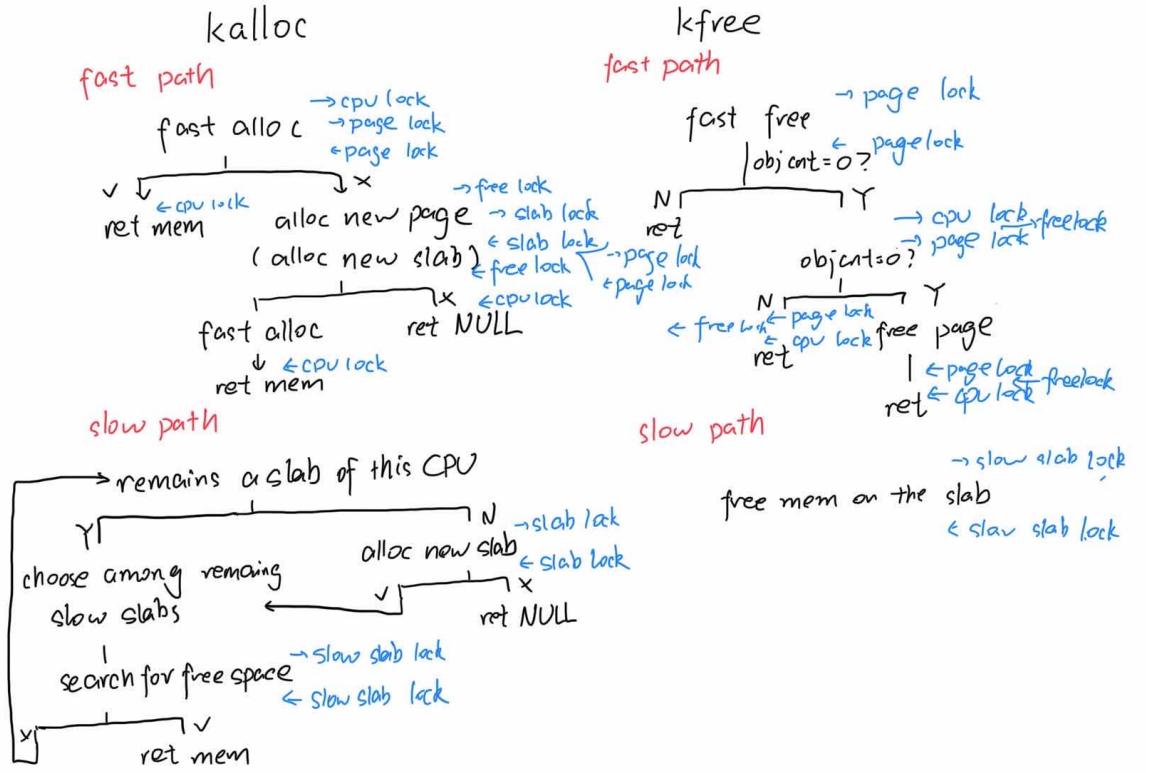
L1 - 物理内存管理

- **数据结构 - 两级Slab**

  划分：由于fast path和slow path需要不同大小的slab，我设计了两级slab。一级slab大小为16 MB，在pmm-init中将物理内存划分为多个一级slab；二级slab大小为32 KB，在fast path中需要将一级slab细分为多个二级slab(称之为page)。slab大小的设计通过宏定义实现，可根据需要修改。

  管理：设定一个全局slab_manager管理所有一级slab，记录slab状态。每个属于slow path的slab通过一个slow_slab_manager管理，manager包含一个map用于记录slab中空间利用情况。fast path通过free_page和fast_slab_manager实现对空闲page及每个CPU的page的管理，每个page则通过初始256 KB的Header管理内部空间的使用。

- **内存的分配与释放**



- **锁的设计与性能优化**

  kalloc和kfree的性能与锁的设计息息相关。我通过如下方式优化程序表现：

  a. kfree在fast path下尽量仅开page的lock，如果发现page上已经没有对象，则对所属的cpu的fast_page_manager上锁，检查page上确实没有对象后删除page
  b. slow path以一级slab为单位上锁，性能大幅提高
  c. kalloc的fast path对fast_page_manager[cpu]的锁细分，仅在第一次fast alloc和添加新page的前后上锁(此方法最终没有采纳，牺牲代码的易读性换取这点性能优化不值得)

- **测试框架**

  我在/kernel/test/中保留了测试代码，分别测试了单核、多核(每个线程只free自己的指针)、多核(线程会互相free指针)的情况。test.c的输出会通过check.py检查，可检测出double allocation、not alligned等问题。可在kernel下make test#x进行测试，#x为测试号。

- **Bug记录**

  a. 分配new page的函数中对一个锁unlock了两次。线程A第一次unlock后，线程B占有了这把锁，却被线程A的第二次unlock释放了，导致线程B和正在等待的线程C可能获得同一个page
  b. 出现了没有被OJ触发的死锁，两个线程可能会等待对方持有的锁而陷入死循环。
  c. 测试框架也是并发程序，线程A free的地址会被另一个线程B的alloc获取。如果A先free后输出记录，可能线程B获取alloc后会在A之前输出记录，从而导致检测程序误判。