L1(pmm)实验报告

slow path: central list

大致实现

所有来自slab的内存请求或者大于32KB的内存请求会通过central list来分配内存, 中央链表只有一把大锁, 且地址按照64KB对齐.

对所有来自slab的内存请求, 直接返回一块大小为64KB的内存, 将中央链表的HEAD放在该块内存的首地址.

对所有大于32KB的内存请求, 会分配"请求内存大小 + 64KB"的内存, 以便64KB的地址对齐. 返回ptr指向请求分配的内存块, 多出来的64KB放在请求的内存块之前, 将中央链表的HEAD放在该64KB的首地址, 虽然有点浪费, 但是简化实现

fast path: slab

大致实现

每个cpu有一把自己的锁, 和一组slab, slab的order从1到log(32KB), 用于分配小于32KB的内存, slab_alloc时不需要加锁, slab_free时需要加锁, 因为可能有多个cpu同时释放属于同一个cpu的内存.

slab同样有个HEAD, 放在central list的HEAD之后.

slab采用引用计数的方式, 当free_cnt = 0时, 请求一个新的slab; 当use_cnt = 0时, 释放该slab.

L2(kmt)实验报告

大致实现

os_trap && os_irq

基本按照讲义实现, 其中handlers以链表的形式存储, 排序在插入的时候就已经做好

task_t

以链表的形式存储, 每个cpu有一个idle task, 用于处理没有ready task的时候的情况; 最初实现的时候, 按照jyy的"嘱咐"先实现一个简易的版本, 因此每个cpu有一个task链表, 不能跨CPU调度, 这时候bug还挺好调,实现后也能过OJ的前4个点; 后来才只有一个tasklist, 允许跨CPU调度了, 并发bug出现, 确实难以调试多了, 但因为已经有一个简易版本的正确性保证, 也比直接实现跨CPU调度要容易定位BUG多

spinlock_t

基本按照jyy的spinlock_t的实现, 但jyy的实现有一个bug, 在push_off()中, 应该先关中断, 再记录当前cpuid, 因为cpu_current()的调用可能会触发中断, 从而导致更改的是中断前的cpu的noff; 因为过于信任jyy(bushi), 调了很久这个bug

sem_t

sem_wait()和sem_signal()基本按照讲义的实现, 但是可能出现"sem_wait中的yield()在被调用前, 就被sem_signal()给标成ready"的bug, 从而sem_wait()的线程还没中断返回就被另一个CPU调度, 解决方案是再sem_wait()中将当前task的状态设置为TOBLOCK, 在中断处理或者sem_signal中, 若该task的状态为TOBLOCK, 则将其状态设置为BLOCK, 若为BLOCK, 则将其状态设置为READY. 这是因为发现一个将要睡眠的睡眠的线程, 只有同时在经历了调度和sem_signal()后才能被标记成READY

调度

kmt_schedule()中有一个很大的bug, 就是在将当前task从RUNNING改为READY后, 那么其他CPU也可以调度当前task, 但是此时可能还在借用当前task的栈进行中断返回, 从而导致栈被破坏, 导致QUMU神秘重启. 一把大锁锁住kmt_schedule是行不通的, 因为中断返回并不是到kmt_schedule()返回就结束了. 解决方案为除了记录当前cpu的current_task, 还记录当前cpu的pre_task, 这一次schedule的cur_task会被标记成TOREADY, 只有在该CPU下一次schedule时才会在kmt_save_context()中将pre_task设置为READY

最后的bug

实现完上述的内容后, 已经可以过OJ了, 但本地的压力测试仍然不能长时间运行. 最后发现是一个上述sem_t和调度的bug的结合版: 如果一个将要睡眠的线程先被schedule()标记为TOBLOCK, 在还没有中断返回时, 又被另一个线程调用sem_signal()给标记为READY, 接着被某一个CPU调度, 就又会发生栈冲突. 解决方案类似, 每个CPU额外记录一个signal_task, 如果一个将要睡眠的线程被schedule()标记为TOBLOCK, 就用signal_task记录下来...(不再赘述).