OS-Lab 实验报告

人工智能学院 丁豪
181220010@smail.nju.edu.cn

Lab1 - pmm

目前实验状态: 过全部easy和3个hard

实验过程:

1）一把大锁保护整个空闲列表 (已经达到最终成果, 能过3个hard)

2）预先给每个cpu分配空间，每人一把小锁，外加一把大锁 (后由第三步取代)

3）每个cpu预分配可动态增长但是不能回收的slab，针对第三个hard test，先做了page_size的slab，后续可以在此基础上扩展其他大小的slab。（本地测试没有问题，但是oj上出现double alloc错误，由于没有更多时间debug，最终没有采用这一部分）

设计精巧的地方:

1）使用双向循环链表加上起始哨兵的结构来管理空闲链表。

2）回收空间的时候根据后方紧邻header的status字段判断其是否是一个空闲分块，以此实现了高效合并空间，省去了每次free都要遍历链表。

```
enum {SPACE_FREE,SPACE_USED}; // 用于space_header的status字段
struct space_header {
  int status;
  size_t size;
  struct space_header* prev;
  struct space_header* next;
};
struct space_header* free_head; // 全局空闲链表头，哨兵
```

3）使用C__builtin_function进行高速位操作，包括slab部分的bitmap相关操作，以及空闲链表中的返回地址对齐计算。

修复的重要BUG:

1）在阶段2的设计过程中曾经碰到过一个bug会导致死锁，当时没有考虑到一个cpu分配的内存可以在另外一个cpu进行回收，因此当时在fast path的设计中根本没有上锁，这样会导致当一个cpu试图free另一个cpu的已分配空间时产生死锁。后来对每个cpu预先分配的区域进行小锁处理，避免了这个问题。

2）slab的设计中有一个定义item_n初始时存在歧义，有时他指的是第几个Item，有时又指的是item的下标，由此会差1。在了解到这一点之后统一了他的含义，排除了这一错误。

Lab2 - kmt

目前实验状态: AC

实验过程:

1）参考xv6，实现带有中断判断处理功能的自旋锁与信号量。在CPU结构体中分别用intr_on参数表示第一把锁锁住前cpu的中断开启状态，hold_locks参数记录当前cpu所持有锁的数量，一旦到0则根据上述intr_on参数恢复中断。

2）实现课堂上演示的有stack race的schelder，但是不进行cpu切换，以此避免stack race，目标通过easy test。最终通过easy和第一个hard。

3）进行类似RCU的改进，解决stack race问题，并使得线程可以在cpu上切换。第二个hard可以通过，但第一个出现thread starvation。

4）通过比较发现，3）的线程调度速度相比2）慢了很多，于是将cpu和线程解除绑定，让cpu直接线性搜索可以调度的线程进行调度，最终AC。

实现精巧的地方:

1）给每个cpu分配一个idle task，在初始状态此task为纯空task，并不分配栈，在第一次中断到来的时候将context保存在此idle task的ctx参数中，以此来“捕获”idle线程，也就是os->run。

2）课堂上讲的stack race情况，是由于线程的新调用者以及原调用者的中断处理程序共用了同一个栈，于是解决的思路就是使得原调用者的中断处理程序保证结束之后才允许调度此线程。实现的方法是给每个线程增加wait参数，并给每个cpu额外设置task_t * last参数来标记上一次调度的线程，当下一次中断到来时，将上次调度的线程释放，并给新线程标记wait，以此避免了两个cpu同时在同一个栈空间上操作的局面。

```
struct task {
    int wait; // 此task是否可以被调度
    const char * name;
    _Context *context; // 初始由_kcontext生成,在栈顶
    void  *stack; // 指向堆栈起始地址指针,需回收
};
struct cpu_local {
    task_t * idle; // 每个cpu专属的idle线程
    task_t * last; // 上一线程
    task_t * current; // 当前线程
    int current_i; // 当前线程在线程列表中的编号
    int intr_on; // 在自旋锁关中断之前是否处于开中断
    int hold_locks; // 当前持有锁的数量
};
```

3）使用了一些宏定义，使得代码更加简洁且易于管理。

```
#define my_cpu  CPU[_cpu()]
#define my_last CPU[_cpu()].last
#define my_current  CPU[_cpu()].current
#define my_idle CPU[_cpu()].idle
#define my_i    CPU[_cpu()].current_i
```

遇到的重要bug：

1）本实验仅一个主要bug。在实验第2）步时，线程调度过程中会出现CPU神秘重启，或者线程卡死不动的情况。通过watch每个线程的_Context * ctx参数，发现他们会在不应该被修改的时候改变，由此推断可能产生了stack race。经过长达数个小时的艰辛debug，在快要崩溃的时候总算找到了元凶，原来是新建线程的时候栈分配出了问题，导致ctx始终处在一个未定义的地址，自然有被corrupt的风险。

```
#define STACKSZ 4096
...
task->stack = pmm->alloc(sizeof(STACKSZ)); // 错误，栈只分配了4Byte!
task->stack = pmm->alloc(STACKSZ); // 正确，分配4kb的栈
```

L3-vfs

目前状态：完全没有完成任何内容

由于时间和能力原因，我放弃了L3。虽然如此，但是我没有抄袭，希望能获得诚信分。