L1、L2 实验报告

姓名：殷天润　　**学号：**171240565

2019 年 5 月 16 日

第一部分　主要架构

A　L1

A.1　alloc 以及 lock

1. 我的 L1 中的工作主要由 alloc.c,common.h 以及 lock.c 组成;
2. 我的自旋锁通过 ncli 数组记录相应的 cpu 进行中断的数量, 在 popcli 中用 assert 来确保 ncli 中的值恒大于 0, 否则就 assert 然后 printf 报错;lock 以及 unlock 都是非常简单的标准实现。(在 L2 中我重新写了一遍, 主要增加了 intena 的判断;)
3. alloc.c 里面主要实现了可以拆分空间的 kalloc 函数以及一个比较简单的 free 函数;
4. kalloc 与 kfree 函数:
   (a) 我通过一个双向链表数组 cpu_head 以及一个结点 unused_space 来维护我的空间分配;unused_space 用来记录未分配的空间位置;
   (b) 我在 kalloc 函数中实现了对空间的优化: 如果申请的空间适当正好小于等于已经开出来的但是之前被 free 的空间 S, 并且 size 本身 > 某一个值, 那么我的算法将拆分 S 为 S_1 =size,S_2 =S-size-sizeof(_node); 限制 size 大小的操作是为了防止最后的空间过于碎片化;
   (c) 我的 kfree 实现了搜索整个链表链然后返回搜索的相应地址的操作;
   (d) 对于申请 size=0 的操作, 我将返回 pm.start, 如果 kfree 的 ptr 为 NULL 那么我将不会进行任何操作;
5. 为了保证程序的正确性 (主要是指针不飘), 我在程序中加入了很多的 assert 来保证; 我在 os.c 中首先进行了申请空间的简单测试, 然后借助, 修改鄢振宇同学的随机测试框架进行了进一步的测试, 调试, 也发现了一些问题;

B L2

B.1 OS

1. 主要的工作是 os->trap,os_on_irq,_handler_length 以及 common.h 中的 handler_list 结构体; 总体思路是 os_on_irq 用来增添 handler_list, 注册函数, 并且保证以 seq 的大小作为 list 排序的依据; 然后 os_trap 在符合情况的时候进行调用;

2. handler_list: seq,event 两个 int 用来记录信息,handler 一个 handler_t 用来保存 handler;

3. os_on_irq: 当 _handler_length 长度为 0 的时候直接在 handler_list 里面注册; 否则遍历 handler_list, 根据 seq 找到可以插入的位置, 并且插入, 注册; 同时都维护 _handler_length 的长度;

4. os_trap: 遍历 handler_list 如果这个 event 是 _EVENT_NULL 或者与 ev.event 一致, 就调用, 用上下文 (_Context) 结构体的 next 保存; 如果 next 存在,ret=next; 执行完遍历之后, 如果 ret 还是 NULL, 那说明有问题 (因为在 init 里面肯定已经注册了两个函数 kmt_swicth 以及 kmt_save)

B.2 KMT

B.2.1 锁

用 am 里面存在的函数改造了 xv6 的 spin_lock, 用全局的 static 的两个 int 数组来记录 intena 以及 ncli;

B.2.2 kmt 的其余内容

1. 数据结构:

   - task
     (a) int status 用来记录状态, 我申请了 3 个 const int 类型的变量: _runningable=1 ,_running=2,_waiting=3;
     (b) name,_Context 类型的 context,_Area 类型的 stack;
     (c) task_t *next, 链表连接下一个;
   - semaphore
     (a) int value;const char * name; 记录相应信息;
     (b) task_t 类型的数组 task_list, 我通过 int 类型的 start,end,MAXSIZE 将这个数组维护成了一个先进先出的队列结构;

2. 全局的变量:

   (a) static task_t * task_head[9] ; static task_t * current_task[9]: 对于每一个 cpu 我都维护了一个链表头和一个当前正在跑的任务;

(b) task_length[9] 用来记录某一个 cpu 对应的 task 链表里面有几个任务;

3. kmt_context_create: 我通过 task_length 数组获得最少任务的 cpu 编号, 然后将 task 存进去, 并且对状态进行更新;

4. kmt_context_save: 正常的存 context; 因为我锁了核并且有 _running 状态, 所以我可以将 _running 在存了之后改成 _runningable 状态;

5. 信号量:

   - kmt_sem_wait: while(sem->value<=0){设置当前任务的状态为 waiting; 更新 task_list 队列; 开锁;_yield(); 关锁}; 然后 sem->value-=1; 这里比较重要的是 sem->value-=1 的顺序; 我之前在 while 之前进行了这个操作, 导致了我的进程会永远的睡着, 没有办法多核多线程, 只能单核的跑单个 echo_task;
   - kmt_sem_signal: 先进先出的从 sem 的 task_list 里面唤醒进程;

6. kmt_context_save: 如果当前没有任务, 我选择不记录这个上下文; 如果当前有任务, 我就用 task 里面的 context 属性记录上下文; 如果现在的状态是 _running, 我会改成 _runningable; 因为我的 cpu 与 task 是锁住的, 并且有三个状态记录, 因此不会出现 stack smach;

7. kmt_context_swicth: 写的比较复杂, 核心思想就是: 如果当前 cpu 没有 current_ 就从 task_head 开始找到第一个 _runningable 的 task 然后选做当前 cpu 的 current_task 并且修改状态; 如果有 current_task, 就从 current_task 的下一个 task 开始找到一个 _runningable 的 task 然后修改 current_task 的状态并且将找到的 task 设置为 current_task;

8. kmt_teardown: 还不是很完善, 不确定能不能通过测试; 我在 task 里面加了一个 int 类型的 alive 变量; 在调用 teardown 的时候, 如果要求释放的 task 的状态是 _runningable 的, 那就删除掉; 如果不是, 就把 alive 标记为 0, 然后在 switch 的时候进行删除;

第二部分　遇到的问题

C　L1

C.1　lock

主要是 spin_lock 中的 bug; 因为对 lock 的不甚了解, 我一共用了三个版本的 spin_lock, 第一个版本仅仅是一个不关中断的玩具实现, 在 tcg 环境下看不出; 什么问题, 在 kvm 中效果很差; 第二个版本参考了开源的 xv6 实现, 可能移植的时候出了一些问题, 一直会报错; 最后我简化了 xv6 实现变成了目前的版本;(L2 中进一步改进了锁)

C.2 alloc

在 alloc 中的主要 bug 是初始化的问题, 指针乱飘导致的 while 中的死循环, 在遇到死循环的时候因为我的 spin_lock 无法确认心理上的正确性; 所以调的比较艰辛

D L2

D.1 笔误型 bug:

将 _cpu() 写成 _cpu, 找了很久

D.2 算法型 bug:

1. 信号量里面 sem->value 要在 yield 之后改变, 不然会多次 wait 导致问题;
2. 在 switch、save 里面的时候遇到了一些设计的问题;

D.3 总结:

这次的 bug 很难定位, 我只能通过尽可能多的输出相关信息来 debug, 因此也耗费了很长时间;

第三部分 潜在的完善以及额外的工作

E L1

我可以将 head 数组变成一个 head 来维护, 在这个基础上就可以在 free 里面合并了;

F L2

F.1 debug 工具

在 debug.h 里面宏定义了 TRACE 以及 log 用于显示信息