OSLab 实验报告

221840206_江思源

L1：物理内存管理(pmm)

代码架构设计

- 使用 buddy system 和 slab 共同实现物理内存管理，很好地实现了结合 workload 进行优化的目的。
  - buddy system 用于大块内存分配：
    大块内存在操作系统中分配的特点是：分配和回收的次数较少，同时一块内存多次使用。所以我们的需求是：不需要很快的分配速度，而需要方便的管理、回收和简单易实现的线程安全。
    buddy system：将内存按照2的幂次划分，每个块的大小为2的幂次，使用全局大锁。在获取（或释放）相应内存块时，只需要找到相应大小的块进行分割（或合并），对效率的影响不大，同时很好地保证了安全性。
  - slab 用于小块内存分配：
    小块内存在操作系统中分配的特点是：较为快速地分配和回收。所以我们的需求是：快速的分配（保证scalability）与回收。
    slab：根据内存大小划分不同 slab，slab 由 cache 管理，cache 持有 slab 链表以及 cache 的锁，slab 持有空闲块链表以及 slab 的锁。在获取（与释放）相应内存块时，只需要找到相应大小的 slab 进行空闲块分配（取空闲块链表头部分配并相应删除）或回收（链接至对应链表的头部），每个部分均有自己的锁，保证了线程安全、并发性和快速的分配。
  - slab 结合 buddy system：
    slab 中的 slab（大小设定为4096，为PAGE_SIZE） 由 buddy system 分配。

```c
struct free_list {
    struct list_head free_list;
    int nr_free;
};

typedef struct buddy_block {
    struct list_head node; // 用于空闲链表的节点
    size_t order;  // 2^order pages(页的阶数)
    int free;  // block是否空闲
    int slab;  // slab分配器，不为空时，表示页面为slab分配器的一部分
} buddy_block_t;

// buddy系统
typedef struct buddy_pool {
#define MIN_ORDER 0 // 2^0 * 4KiB = 4 KiB
#define MAX_ORDER 12 // 2^12 * 4KiB = 16 MiB
    struct free_list free_lists[MAX_ORDER + 1]; // 空闲链表（每个对应相应order）
    lock_t pool_lock[MAX_ORDER + 1]; // 保护伙伴系统的锁
    void *pool_meta_data; // 伙伴系统的元数据
    void *pool_start_addr; // 伙伴系统的起始地址
    void *pool_end_addr; // 伙伴系统的终止地址
} buddy_pool_t;
```

```c
// 空闲块链表
typedef struct object {
    struct object *next;
} object_t;

typedef struct slab {
    struct slab *next;        // 指向下一个slab
    object_t *free_objects; // 指向空闲对象链表
    size_t num_free; // 空闲对象数量
    lock_t lock; // 用于保护该slab的锁
    size_t size; // 每个对象的大小
} slab_t;

typedef struct cache {
    slab_t *slabs; // 指向slab链表
    size_t object_size; // 每个对象的大小
    lock_t cache_lock; // 用于保护该cache的锁
} cache_t;
```

精巧的实现

- 位运算技巧：
  - buddy system 中的合并获取相邻块的地址：通过位运算（addr ^ (1 << (block->order + PAGE_SHIFT))），获取相邻块的地址，再判断是否合并。
  - slab free：通过空闲块地址位运算（后12位置0）获取 slab 地址
- 通过 cache_t 来管理 slab：
  - 实现多级锁：cache_t 持有 cache 的锁，slab 持有 slab 的锁，在实现的时候思路更加清晰，并且保证了scalability。

印象深刻的bug

- 指针类型转换：有时候对一个地址进行不同的类型转换，再进行自加（减）运算会有不同的结果，一般会加（减）自己的类型大小。调试的时候让我对各种指针转换理解加深很多了。
- 指针没有对齐2^i：单纯的变量名写错了，导致 bug 找了好久，所以写代码使用更良好的命名风格很重要。

L2：多线程管理(kmt)

代码架构设计

- MODULE(os)
  - os_trap: 用作中断处理函数，其中调用handler函数处理中断，而handler由一个链表维护，在注册处理函数的时候使用冒泡排序来进行插入。（注意：abstract machine在调用中断处理程序，也即os_trap时会关中断，在写代码的时候需要注意）
  - os_on_irq: 用作中断注册函数，将handler注册到对应的中断号上。
- MODULE(kmt)
  - spinlock_t
    - 参考xv6自旋锁实现，保证了原子性，并检查是否有死锁（事实上这个检查给我debug造成了不少困扰...）。
  - task_t
    - 关键成员：status（用来标注线程的状态），stack（由kcontext在上面创建上下文），fense（对栈完整性的检查），数据结构和kmt.c中的全局变量

```
struct task {
    const char *name;
    int id;
    int cpu_id; // debug need
    task_status_t status;
    Context *context; // 指针
    uint32_t stack_fense_s[STACK_GUARD_SIZE];
    uint8_t stack[STACK_SIZE];
    uint32_t stack_fense_e[STACK_GUARD_SIZE];
};
```

```
static spinlock_t task_lk_spin = spinlock_init("task_lk");
//--------protected in task_lk---------
static task_t *tasks[MAX_TASK_NUM]; // all tasks
static int total_task_num = 0;
static task_t *currents[MAX_CPU_NUM]; // 当前任务
#define current currents[cpu_current()]
//---------------------------------------
```

    - kmt_create: 创建线程，初始化线程的上下文，将线程加入到 tasks[MAX_TASK_NUM] 中。
    - kmt_teardown: 销毁线程，将线程从 tasks[MAX_TASK_NUM] 中删除并设置 tasks[task->id] = NULL。
    - 两个基础处理函数的实现。
      - kmt_context_save: 保存当前线程的上下文，注意上全局的保护 task 的大锁。
      - kmt_schedule: 调度线程，注意上全局的保护 task 的大锁。具体实现方法在“精巧的实现”这一板块中详细描述！
  - sem_t
    - 关键成员：spinlock（保证本信号量代码中的原子性），value, queue（这个实现一定要小心...）

    - 具体实现就使用spinlock保证原子性，queue来维护等待队列，value来维护信号量的值。对原理的理解就放到“精巧的实现”这一板块中阐述。

精巧的实现

- Context *kmt_schedule(Event ev, Context *ctx);
    1. cpu上空转的任务：idle

```
static task_t idle[MAX_CPU_NUM];        // cpu 上空转的任务
```

    - cpu上执行中断处理程序时，如果没有其他任务可以执行，就会执行idle任务，这样可以保证cpu不会卡死，也给其他任务提供了执行的机会。
    2. task 状态的转换

```
task_t *currents[MAC_CPU_NUM];          // 当前正在执行的任务
task_t *tasks[MAX_TASK_NUM];            // 所有的任务
```

    - 通过current来记录当前cpu正在执行的任务，通过tasks来记录所有的任务。
      task的状态有：RUNNING, RUNNABLE, BLOCKED
        - RUNNING：当前任务正在执行，仅可能有cpu上的一个任务处于这个状态
        - RUNNABLE：当前任务可以被调度执行，但是没有被调度执行（在切换前会将状态为RUNNING的current任务设置为RUNNABLE）
        - BLOCKED：仅由信号量的P操作引起，当前任务被阻塞，不会被调度执行，并且只有V操作可以设置BLOCKED的任务为RUNNABLE
        - 由此完成了任务状态的转换，保证了任务的正常调度。
    3. 信号量的实现
    - 先设置初始value，用value来表征某一个资源的数量
    - P操作：如果value > 0，value--，否则将当前任务设置为BLOCKED，加入到等待队列中
    - V操作：如果等待队列不为空，将等待队列中的第一个任务设置为RUNNABLE，否则value++
    - 等待队列就像一个缓存区，当value不够用时，将任务加入到等待队列中，当value有剩余时，将等待队列中的任务设置为RUNNABLE，这样就实现了信号量对于任务的控制并保证不会丢失任务（由此实现了同步）

印象深刻的bug

- 访问空指针：在用链表实现队列的时候，我初始化的时候没有传二重指针，从而导致并没有修改链表头指针，而是修改了一个局部变量，从而导致访问空指针（undefined behavior），然后undefined behavior在 abstract machine 上会导致调用中断处理程序，所以我在中断处理程序上获得锁的行为会直接导致死锁，但是我的实现按照常理来说并不会死锁...所以这个bug找了挺久的，还因此重构过代码。
- 其他bug其实都是没想清楚的问题，只要仔细画出task的状态转换图就可以很快解决了，毕竟 everything is a state machine（