实验报告

学号	姓名	院系
221220046	宋承柏	计算机科学与技术系

L1：物理内存管理 (pmm)

架构设计

pmm的实现总体上使用了Fast Path + Slow Path的方式：首先将堆区划分为若干固定大小的页面（本实现使用了64KiB作为页面大小）。

- 对于小内存分配，走fast path。每个CPU都拥有一些专属于自己的page，每个page都使用Slab的方式，划分为固定大小的内存单元，其中第一个内存单元被用于存储page的元数据。page使用free_list来跟踪页内可用的内存单元。在本实现中，内存单元的大小被分为8个等级（32B ~ 4096B），因此每个CPU需要维护8条page链表。小内存分配时，在线程本地（页面内）即可完成分配。
- 对于大内存分配，走slow path。获取全局锁，从全局的空闲链表中找到一段连续的pages进行分配。

印象深刻的Bug

Double allocation

最初我没有使用Slab的方式组织页面，而是需要多少大小的内存，就从全局的内存中“切”一块下来。这样造成每个页面的大小都各不相同，十分混乱，出现Double allocation、Hang等错误提示。后来我重构了代码，使用了Fast,Slow Path + Slab的方式，就不再出现这一bug了。

精巧的实现

使用Fast，Slow Path + Slab不仅可以更方便地满足地址对齐要求，还很好地降低了实现的复杂性。这给我的经验是，对于复杂度较高的程序，使用System的方法比精巧但是复杂的数据结构更有效。

不足之处

- 我的代码中存在代码克隆，增加了维护和改进的复杂性。引入代码克隆是因为我最初使用了低效的命名方式，我对pagelist中的每个不同等级的page都使用了不同的名称，例如`pages_512B`。而事实上使用二维数组即可。若有时间再对此进行改进。
- 无法支持大内存的free。在我的实现中，若对大内存执行free会导致OJ出现too slow错误。这可能是因为我对大小内存都使用了串行的链表来组织所有页面，在寻找需要free的大内存时需要遍历所有页面，因此效率过低。因此虽然能通过OJ，但是在实际使用场景中若运行充分长时间，堆区内存将耗尽而引起崩溃。若有时间再对此进行改进。

L2：内核线程管理 (kmt)

架构设计

kmt的实现主要分为三个部分：自旋锁、信号量、线程管理。

自旋锁

自旋锁的实现直接参考了xv6代码中的实现方式：

```
typedef struct spinlock
{
    const char *name;
    int status;
    cpu_intr_t *cpu;
} spinlock_t;
```

信号量

为了实现可睡眠信号量，另外定义了一个结构`wl_node`表示等待队列，这是因为总的线程池也是使用链表实现的，为了将每个信号量上的等待队列与总的线程池区分开，额外定义了这一结构体。

```
typedef struct wl_node
{
    task_t* task;
    struct wl_node* nxt;
    struct wl_node* pre;
} wl_node_t;
```

```
7
8  typedef struct semaphore
9  {
10     int count;
11     const char* name;
12     spinlock_t lock;
13     wl_node_t* wait_list;
14 } sem_t;
```

线程管理

对于kmt模块的初始化、线程的创建、删除，提供了`kmt_init`，`kmt_create`，`kmt_teardown`接口，这一部分比较简单。需要注意的是，要在init中完成线程池的锁的初始化，并为每个cpu都创建一个idle线程，当该CPU没有可用线程可以执行时，就返回idle空转。

而困难的部分在于，在os_trap中执行的两个中断处理程序`kmt_context_save`和`kmt_schedule`。

如上所述，我的实现使用了一个总的线程池（链表），除idle线程外，每个线程都作为一个节点。

当发生时钟中断或执行yield时，进入os_trap执行，os_trap按序调用已注册的中断处理程序。首先，`context_save`被调用，保存上下文至进程结构体中；最后，调用schedule，从线程池中获取可用的线程，如获取成功，就返回其上下文，否则返回idle的上下文。

印象深刻的Bug

栈上的数据竞争

由于在之前的课程上已经做出了提示，因此对于该bug我在开始做实验之前就已经知道其存在。虽然如此，找到一个合理的解决办法依然花了我相当多的时间。具体而言，在中断发生后，AM首先会将当前上下文压栈，然后调用os_trap，os_trap返回新的上下文后，再由AM将其恢复到栈上。由于os_trap并不是整个中断处理过程，因此在os_trap中加一把大锁是不能避免数据竞争的。

`push_off`中的关中断时间

在移植到AM的`spinlock`代码的`push_off`函数中，获取`mycpu`是在关中断之前，而这是有问题的，因为在获取mycpu、关中断之前，若发生中断，那么后面操作的cpu就不再是当前执行该线程的cpu了。该bug直接导致OJ启动多核时触发assert。由于最初没有对这部分代码的正确性有过怀疑，因此找到这个bug花了很久。这给我的经验是，机器永远是对的，而别人的代码，即使以前能正常工作，也不一定是对的。

精巧的实现

为了解决跨核调度时栈上的数据竞争，我尝试了很多办法，最终使用了延迟调度的方式。具体而言，使用一个last数组记录每个CPU上运行的上一个进程，在每次中断进入`context_save`时，才将last标记为可运行。这样就能保证不会发生在中断处理程序还没返回之前，某个线程就被别的CPU调度的情况。

为了实现延迟，我最初的解决办法是每个线程用一把锁，在shedule时，尝试对选中的线程获取锁。在save时，unlock上一个线程的锁。该方法确实解决了数据竞争问题，但是在我的实现中，无论使用何种调度策略，使用每个线程一把锁的方式会造成OJ第五个测试点CPU starvation。其原因不得而知，经过漫长的调试，最终我弃用了线程锁，改为schedule时标记线程状态为RUNNING，save时标记last线程状态为RUNNABLE。为了保持简单，调度策略使用了随机调度。该方法最终通过了OJ。

值得一提的是，为了debug，有几次我没有对跨核调度的线程状态做特别处理，因此在本地测试时发生了stack smash，但是提交后却通过了OJ，因此我认为OJ的测试可能不够全面。

漫长的调试过程给我的启示是，为了写出真正有效的程序，先从一个简单但是正确的实现开始，逐步细化实现从而得到一个能正常工作的程序，要比一开始就使用复杂的策略更有效。而在调试过程中，对每一个部分进行单元测试是很重要的，单元测试能够确保某一个模块不出现问题，从而更快地定位bug。