OSLAB 1 实验报告

161220049 黄奕诚

一、实验整体描述

完成要求的所有实现，共用 10 小时左右（含 Debug 时间）。

二、设计思路

考虑到自己实现的功能与讲义要求基本一致，没有太跳，所以不再赘述各个功能的具体实现，只是简要地阐明一下设计思路、Debug 技巧、几个 Bug 的查改经历和个人的感想。

pmm 部分的代码可以直接从 MINILAB5 的 malloc 与 free 实现中移植过来。因为是否2^k对齐对后续 kmt 的实现影响不大，所以对线程调度、自旋锁和信号量相对熟悉的我计划先实现 kmt 部分，最后进行2^k对齐。

1.自旋锁

在实现 kmt_create 和 kmt_teardown 之前有必要先实现自旋锁，而这个在课上和书上都有详细的阐述，主要要注意如下问题：

a) 在 unlock 一个锁时，什么条件下才能开中断？

b) 在重复开/关同一个锁时，应该怎样处理？

通过记录每一个锁的 locked 状态和定义一个全局的 lock_cnt 变量可以有效地解决上述问题，此外还需一个 intr_ready 变量标识是否符合开中断的条件。

2.线程创建、回收和调度

在抉择使用链表维护多个线程还是开一个静态数组 thread_pool 管理多个线程时，我选择了前者（两者实现难度都不大）。每一个线程我维护了它是否可执行（不在 sleep）的状态，记为 runnable、它对应的信号量 waiting_sem、特殊标记 tid、寄存器现场 tf 和堆栈 stack（Area 结构，同时维护 start 和 end），加之其下一个线程 next。每当创建一个线程时，将其添加到链表头；每当回收一个线程时，将其从链表中删除。实现 kmt_schedule 时，采用 robin-round 方法，采用循环链表的方式依次调度到下一个 thread 即可。所以实现这一部分的思路并不难。

3.信号量

主要参考了 OS 讲义 lec12 的实现，只是没有实现条件变量，将其用触发中断和设置 runnable 的方式替代，这样也更加简洁。

4.内存分配与释放

采用了空闲链表的方法，维护每一个空闲块的起始地址、大小等信息。在设置2^k对齐时，对原先代码进行了如下修改：

a) 当没有空闲块可以满足当前所需的 size 时，将 brk 向后拓展 2 倍 size 的大小，如此必然可以容纳2^k的所需大小，因为可以证明假若 k 是满足$2^k \geq size$的最小整数，则必然有$2^k < 2 * size$

b) 当获取到参数 size 时，直接将其对齐到2^k

c) 当找到一个足够容纳当前所需大小的空闲块时，将返回地址对齐到2^k即可，这点在 a)中可以得到证明

如此即可为后面的虚拟地址空间进一步实现提供便利。

三、DEBUG 技巧

采用实验讲义提供的建议，在/include/debug.h 中定义了 assert、Log 和 TRACE 三个宏。如下所示：

```
#include "os.h"

#define assert(cond) \
    do { \
      if (!(cond)) { \
        printf("\033[31mAssertion fail at %s:%d\n\033[0m", __FILE__, __LINE__); \
        _halt(1); \
      } \
    } while (0)

#define TRACEME
#ifdef TRACEME
  #define TRACE_ENTRY/* \
    printf("\033[34m[trace]\033[0m %s:entry\n", __func__)*/
  #define TRACE_EXIT/* \
    printf("\033[34m[trace]\033[0m %s:exit\n", __func__)*/
#else
  #define TRACE_ENTRY ((void)0)
  #define TRACE_EXIT ((void)0)
#endif

#define Log(format, ...)/* \
  do { \
    printf("\33[1;34m[%s,%d,%s] " format "\33[0m\n", \
        __FILE__, __LINE__, __func__, ## __VA_ARGS__); \
  } while (0)*/
```

在不需要调试信息时，只需要像图中一样注释即可。在打印信息时添加适当的颜色，可以提升 debug 体验，并将其和正常的 printf 区分开来。

此外，我为三个阶段（pmm、thread、sem）编写了测试代码，并利用测试代码找到了好几处隐蔽的 bug。但是现在仍然不太放心，因为我并不知道如何编写那种很牛逼很有用的测试代码【希望老师有时间能指导一下】。测试代码放在/src/test.c 中。

四、DEBUG 经历举例

1.忘记-O2 编译优化选项，使得在信号量 WAIT 中的一个 WHILE“死循环”被干掉

```
static void kmt_sem_wait(sem_t *sem){
    kmt_spin_lock(&sem_lock);
    Log("%s: sem_count: 0x%x",sem->name, sem->count);
    while (sem->count == 0){
        current_thread->runnable = 0;
        current_thread->waiting_sem = sem;
        Log("%s: 0x%x",current_thread->waiting_sem->name, current_thread->waiting_sem);
        kmt_spin_unlock(&sem_lock);
//        while (current_thread->runnable == 0);
                _yield();
        kmt_spin_lock(&sem_lock);
    }
    sem->count --;
    kmt_spin_unlock(&sem_lock);
}
```

起初我写的是 while (current_thread->runnable == 0); 然后发现这玩意好像不需要唤醒就能跑下去。经过大量的 Log 输出，还是不太明白问题在哪。这里一个大佬说他用_yield 就能跑，我也试了试，然后就成功了……但是用 while 在语义上没问题哎，就让那位大佬也用 while 试了试，发现跟我一样的错误……后来把-O2 删掉，突然正常。好的，编译优化实锤了，下次一定不忘记加 volatile 233333

2.写代码时仿佛灵魂出窍，链表遍历都能写飘

当只创建两个线程的时候，迷之发现永远进不了第二个线程，然后用 Log 打信息发现图中的 next_thread 全写成了 current_thread……一个人永远无法理解以前的自己到底在想写什么，这是很气人的。

```
while (1){
    if (next_thread != NULL && next_thread->runnable == 1)
        break;
    if (next_thread->next != NULL)
        next_thread = next_thread->next;
    else
        next_thread = head;
}
```

3.在修改 2^K 对齐时，发现原先能跑的线程测试代码出了 ERROR

经过 BUG 的排除，发现是在分配新的内存时，没有完全把原先分配的 size 改成 2*size，这导致了每个空闲块的 size 信息和其实际的 size 不一致。这个问题看上去很白痴，其实也警醒我要注意变量的一致性，将要分配的大小新定义一个 alloc_size = 2 * size，就不必改动每一个 size 了（也是一个很低级的 C 语言技巧，可惜我现在还会在这里面栽跟头）

五、个人感想

对于这次实验，个人认为总体的思路设计难度不大，但具体实现时容易栽的坑不少，所以有必要腾出大块的时间进行 debug 和 test。另外，我和其他几位同学都觉得单元测试很重要，但在之前没有很多训练，想请老师花大约一节课的时间指导一下。