

OS Lab实验报告

181240035

刘春旭

L1: 多处理器上的物理内存管理

实验进度:

- 完成了 `kalloc` 和 `malloc`，支持最多8个线程4KB以下的快速并行内存分配；
- 个人觉得本次实验有一些值得写的东西，故实验报告多于2页纸，请见谅。

拖延真的不是好习惯。——鲁迅（没说过这话）

1. 代码思路:

本实验的基本思路就是处理两种类型的内存分配，slow path和fast path。每个线程都保留自己的链表头，这个页面头连接着不同大小的slab页面：

```
typedef struct __head{
    page_t *_32;
    ...
    page_t *_4096;
}list_head;
```

其中，寻找对应slab大小的函数为位运算（round up to the nearest power of 2）：

```
size_t ALIGN(size_t v){
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    v++;
    return v;
}
```

然后根据当前线程的序号、大小的类型就可以从链表的头部找到对应的slab页面的头部，然后进行内存分配。思路基本同课上讲解。

2. BUG之路

什么吗，这个实验也没有多难嘛...（希望之花.mp3）

回过头来看确实这个实验的每一步，蒋老师都有过贴心的提示。所以其实这个实验最值得提的就是debug之路了。

- 刚开始的时候没有意识到可以把指针强制类型转换成结构体指针

我真傻，真的。刚开始做的时候觉得，我该怎么把页面写到堆区呢？我当时真的觉得只有malloc才能帮我实现把数据分配到堆区，但是现在要自己实现malloc，我人就傻了。直到ddl前三天的一个午后，我睡醒惊觉，把 `void *` 转化成 `page_t *` 之后不就可以随意读写结构体对应位置的值了吗。（午睡有益身心健康）

- 栈溢出

这个真的是一个大问题，由于上面第一点提到的问题，我觉得只能把头部和数据区分开，这样不也可以正常返回堆区的地址吗，所以我就把头部声明在栈区、静态数据区了（当时我没意识到这个想法的危害）。于是随着页面越来越多...某个位置的值就一直写不上了。

```

Hello World from CPU #0
-----
alloc(2049): calling: alloc(4096)
PAGE: 00210000: bitmap[0]:0
alloc(2049): 00210000 ← 分配第一个(kalloc(2049)), bitmap[0]=0, 返回对应地址
=====
alloc(2050): calling: alloc(4096)
PAGE: 00210000: bitmap[0]:1
PAGE: 00210000: bitmap[1]:0
alloc(2050): 00211000 ← 分配第二个, bitmap[1]=0, 返回对应地址
=====
alloc(2051): calling: alloc(4096)
PAGE: 00210000: bitmap[0]:1
PAGE: 00210000: bitmap[1]:1
alloc(2051): 00212000 ← 分配第三个时, 第一个页面(共8096B)已经分配出去了, 所以申请了一个新的页面(0x212000), 将其bitmap[0]置1, 并返回其地址
=====
alloc(2052): calling: alloc(4096)
PAGE: 00210000: bitmap[0]:1
PAGE: 00210000: bitmap[1]:1
PAGE: 00212000: bitmap[0]:0 ← 但是分配第四个时, 它认为新的页面的bitmap[0]=0, 所以返回了一个和上面申请相同的地址。
alloc(2052): 00212000
=====
alloc(2053): calling: alloc(4096)
PAGE: 00210000: bitmap[0]:1
PAGE: 00210000: bitmap[1]:1
PAGE: 00212000: bitmap[0]:1
PAGE: 00212000: bitmap[1]:0
alloc(2053): 00213000
=====

```

```

1 #include <./header.h>
2
3
4 // #define ALIGN
5 // #define ALIGN
6
7 size_t ALIGN(size_t)
8
9 static void os_init()
10 {
11     pmm->init();
12 }
13
14 static void test()
15 {
16     init_cache();
17     for (int i = 20; i < 100; i++)
18     {
19         printf("alloc %d\n", i);
20         //pmm->free(p);
21         printf("=====");
22     }
23 }
24
25
26
27 static void os_run()
28 {
29     for (const char *s = "Hello World"; s; s++)
30         _putc(*s == ' ');
31     test();
32     while (1); //keep running
33 }

```

当时我：

- 怀疑可能编译器优化掉了这条给bitmap赋值的指令，故我换成了汇编代码：

```
__asm__ __volatile__ ("mov $1, %0" : : "r" (new_#blk_#sz.bitmap[0]) : );
```

注释掉printf后依然有问题。

- 怀疑是编译器乱序执行指令的锅：

但是我尝试在这条赋值的后面加入了 `barrier()` 之后

```
#define barrier() __asm__ __volatile__ ("" : : "memory")
```

所以我觉得可能是在这条指令后面调用了函数的话，这条指令的执行顺序就在某种程度上被保证了吧。

但是这其实是栈溢出的原因。首先，我原先在函数内部分配的变量是分配到栈上的，函数结束之后是会被销毁的；其次实验讲义中要求过不要把过多的内容分配到静态数据区。

解决方案：意识到第一点后重写，其实还蛮快的，大概一个下午就重写好了。

- 写了很多宏

写了大概100多行的宏...很难受，这也是我下决心重写代码的原因之一。刚开始写宏的时候很开心，觉得很方便，但后来就越写越多...

解决方案：可以先预留出足够的空间，足够容纳下无论哪种大小的slab的bitmap，然后就可以统一定义了。另外，bitmap也不够bit，我定义成了bytemap。反正要求对齐的话，是肯定要浪费一部分内存的。

- *fast path*的数据竞争？

在实验的后期，hard test后几个点轮流出问题。接近崩溃的时候，和同学讨论发现，有很多同学都在bitmap上加了锁。确实，实验的讲义上也给每个页面都加了锁。但是我认为fast path的内存都已经是线程独享的了，无论在哪个线程free，我再分配的时候也不会出现数据竞争的问题，因此不用给bitmap加锁。可能出现数据竞争的位置只有可能是sbrk的时候，因为毕竟只有堆区才是数据共享区。所以后来我发现，是自己的unlock没有用原子指令.....

解决方案：unlock改用原子指令，且只在sbrk处加一把大锁，未在bitmap上加锁。

L2：多处理器内核上的线程管理

这个实验开始时看手册根本没看懂要干什么，但是经过整理：我们要在这个实验中实现kmt模块（kernel multithreading）。我们在L1实现的多线程同时分配只是可以在多个cpu上不停的进行内存的 `alloc` 和 `free`，但是现在我们的CPU还是不支持中断的。中断是为了让操作系统拿到线程的控制权而产生的，

实验进度：完成实验中 `kmt` 模块

1. 代码思路：

我们一共需要写的内容其实就是：

- `task_t` 定义（需自己参透）

这里我的 `task_t` 中的关键结构其实和 `thread-os.c` 及其他示例代码中差不多，外加很多调试结构（以及后来加上的金丝雀，真的好用）。注意这里的结构体大小不能大于L1中的上限（我这里是4096B），否则分配会出问题。

- `kmt` 模块：一组自旋锁函数、一组信号量函数、一组创建线程函数；

- 自旋锁函数：完全按照书上的内容写就可以，注意在lock时关中断，unlock时开中断即可。
intr在我的实现中被储存在了锁里，用于unlock时将intr还原。

- 信号量函数：这个函数会卡住easy-test的第二个点，也是比较痛苦的一个点。当我一开始认为这个函数就像自旋锁一样，可以独立于线程模块实现时，我发现我错了。这里必须先有线程的设计，才能去实现信号量函数。否则试想：`signal` 函数叫醒线程的设计在没有线程的情况下，该如何实现呢？

`sem_wait()`：被调用时，向线程中记录信号量信息，将线程的状态改成SLEEPING，随后yield出去即可（也可以不yield）。

`sem_signal()`：被调用时，在线程列表中比对信号量信息，将对应的线程改成 ZOMBIE 状态，在下次被调度时改回 RUNNABLE 状态，才能进行调度（希望能减少一部分的stack race情况）

- `os` 模块：一组 `os_trap` 函数；这里基本和讲义上给出的代码相同，我分成了一个 `os_trap()` 函数和一个 `os_on_irq()` 函数。这里需要通过研究AM代码，了解传入 `_Context` `*os_trap(_Event ev, _Context *context)` 中参数的含义。其中，中断事件 `ev` 真的非常便于调试。

思路大体上是：遇到中断 `os_trap()` --->调度中断处理函数 `save_context()`（将context保存到目前进程的栈里）---->调度中断处理函数 `schedule()`（选择下一个该调度的进程）---->返回下一个进程的Context

2. BUGS

1. 虚拟机神秘重启

这个原因确实有很多，所以只能用最系统的方式来找到bug。使用课上讲过的gdb以及qemu的 `-D -d` 选项，基本可以定位到出错的位置。其实基本上出错都会处在栈上，所以仔细检查一遍栈的空间问题即可。我是因为给一个结构体分配内存时，只分配了一个指针的空间，导致返回的指针飘飞。当然，在stack race的时候也经常会出现这样的问题。

2. stack race

这是一个大问题。初版的（错误）解决方案就是返回一个复制的 `_Context`。但是这样并不能保证栈上的东西还是对的。所以还需要复制一个栈，然后返回一个正确的、rsp指针指向复制过的栈的 `_Context` 指针。但是出现了新的bug。在使用这种类似read-copy-update的方法时需要注意很多细节上的东西。所以目标暂且转变为尽可能减少stack race，我是通过限制线程调度的次数（不能连续多次）和持续时间（不能过短）来尽量限制的。

- 3. 自己写宏来调试的体验很好，可以一键关闭/开启。并且这个实验中各种各样的assertion能帮很大忙，可以说这个实验帮助我优化了我的调试过程。