L1实验报告

一，架构设计

以4KB（一页）为界，分为大内存与小内存分配。

对于小内存分配，采用了类似slab系统的设计。每一类的小内存（32B,64B,128B,256B,512B,1KB,2KB)都有一个大链表，大链表的每一项都管理者大小为4KB的页，每一页里管理着一个小内存块空闲链表。大链表的项放在4KB大小的页的开头（**SLAB STICK**），里面记录着当前页表可分配的小内存块个数，小内存块空闲链表的链表头，以及一个锁，用来保护4KB页内的内存分配释放操作。

对于多线程来说，每一个线程都有一个这样的小内存管理系统。当当前线程希望alloc一个小内存的时候，它会向自己的小内存管理系统里要一块小内存块，也就是找到对应大小的页，在页里面的小链表里删去一块小内存，将地址返回。这是fast path。

当当前线程的小内存分配系统里分配不出要求大小的小内存块时，它会向大内存分配系统要一块大小为4KB的页，初始化为对应大小的小内存分配管理页，插入系统里面。再从新加入的系统里分出一块小内存地址返回。这是slow path。

对于小内存释放，由于分配的小内存都是在对齐到4KB大小的页里分出来的，所以只要将小内存地址低12位清零，这样就可以获取对应页的**SLAB STICK**，获取此页的锁，将该小内存地址作为小链表的一项插入即可。

对于大内存分配，采用了类似bma系统的设计。由于最大的内存分配是16MB，只需要将大内存划分为开头对齐至16MB的块，每一块用一棵线段树管理即可。每一棵线段树有一个锁保护树的分配释放操作。由于有多棵树，而spin_lock只会在一棵树上不停的试图获取锁，所以我自己实现了一个try_lock：

```
inline bool try_lock(int *lk) {
  return !atomic_xchg(lk, 1);
}
```

每次获取锁的时候try一下，失败就换一棵树，这样可以有效利用多棵树。这也是slow path。

二，精巧的实现

对于大内存的线段树，我的树的最小节点是4KB，一共是13类节点，除了4KB大小的节点，节点可能的状态还有被占用**FULL_USED**以及被完全分裂**GET_DEPARTED**两种，一共15种可能状态，用1字节的char类型就足够了。所以对于512MB大小的内存，我其实只需要64KB大小的内存就可以进行管理。

三，令我印象深刻的bug

我一开始的报错是 double alloc，由于我 slab 编写的很简洁，而 bma 很臃肿，而且在我的测试框架上 thread sanitizer 一直报大内存那里有数据竞争，所以我一直认为是 bma 系统实现有 bug 。但是后来发现我给 bma 系统全局加锁后依旧报数据竞争，我就有点沮丧了，因为很显然 thread sanitizer 假阳性了，这也意味着我之前 debug 方向并没有依据。直到我决定在测试框架里把自己实现的自旋锁换成 pthread 提供的互斥锁，thread sanitizer 才停止假阳性。这时我已经发现大内存分配似乎没有问题，而由于我对自己 slab 实现的自信（简洁，结构清楚），导致我一直没有尝试 de slab 系统的 bug。正好换成互斥锁之后，pthread 给我报了一个试图解锁未上锁的锁的 assert error，我才决定去 slab 那里看一下。结果发现了一个遍历寻找未上锁且有空闲的 slab 页的bug。

```
        if (free_space->current_slab_free_space == 0) {
            free_space = (SLAB_STICK*)free_space->next_slab_stick;
#ifndef TEST
            spin_unlock(&free_space->slab_lock);
#else
            mutex_unlock(&free_space->slab_lock);
#endif
            continue;
        }
```

这是我原来的设计，因为要读取该页上的数据，所以此页要保持上锁状态进行链表遍历，然而，这就导致下面的解锁其实是解的后一个页面的锁。

修完这个 bug 后依旧报 double alloc 的错误。但是 thread sanitizer 依旧什么反应都没有（与之前疯狂假阳性作为对比，哈哈）。所以我采取的 debug 措施是在分配的地址里写入一个“金丝雀”：每次 alloc 时 assert 地址里面的“金丝雀”变量为0，然后加1；每次 free 的时候 assert 地址里面的“金丝雀”变量为1，然后减1。然后我就体会到了测试框架的威力：

我一开始还是怀疑大内存分配有问题，由于报 double alloc 的地方是 normal workload，所以我一开始只让测试框架随机获取释放大小为4KB的块，但是跑了半小时都没问题（开的线程数是6）。所以我不得不相信是我的 slab 出问题了。果然，换成小内存分配的时候几秒钟就报了 alloc 的“金丝雀” assert error。但是当我让测试框架只随机分配不释放的时候，slab 又没有问题了。所以很明显，bug在 slab释放的时候。幸好我的 slab free 非常简单，除去 assert 语句和 log 一共7行，所以一看就发现了并发 bug：

```
        slab_block->next_free_slab_block = slab_stick->current_slab_free_block_list;
#ifndef TEST
    spin_lock(&slab_stick->slab_lock);
#else
    mutex_lock(&slab_stick->slab_lock);
#endif
    slab_stick->current_slab_free_block_list = (uintptr_t)slab_block;
    slab_stick->current_slab_free_space += 1;
#ifndef TEST
    spin_unlock(&slab_stick->slab_lock);
#else
    mutex_unlock(&slab_stick->slab_lock);
#endif
#if defined TEST && LOG
    LEAVE_FUNC();
#endif
```

我锁加漏了一行：）。

修改完就过了。至此L1花费1个月。

L2实验报告

一，架构设计

使用数组存储 task 列表，链表存储 handler 。在信号量和schedule里面，使用随机数随机唤醒或者分配task。task 由 kcreate 创建，由 kteardown 回收。task 的分配则是在 schedule 和 context save 里两个 handler 完成。

二，精巧的实现

自旋锁完全仿照 xv6 的设计进行开关中断。task 的元数据里面有两个变量控制 task 的调度，一个是 is_running，另一个是 block，任意一个值为 true 都不可以被 schedule 调度。其中 block 由信号量控制，如果没有资源就将其放入 信号量自身的 task list 里面，将 block 设为 true，等待唤醒。如果有资源就随机唤醒一个，将 block 设为 false 并将其移出信号量的 task list。对于 schedule，每个 cpu 都有一个 idle task，防止 cpu 没有合适的 task 运行。在获取合适的 task 之后，就将它的 is_running 设为 true。为了使发生中断的 task 的中断完全退栈之后再给别的 cpu 运行，我给每个 cpu 增加了一个 save task，其中保存的是 cpu 上一个运行的 task 编号。每当 cpu 当前的 task 发生中断，都得将当前的 context 保存在当前的 task 里面，然后将当前的 task 设为 save task，因为接下来要进行 task 的切换。在新的 save tast 覆盖旧的 task 之前， 将 save tast 的 is_running 设为 false，给别的 cpu 运行。在这个时候才修改 task 的 is_running 位是因为如果在 schedule 里面一获取到新 task 就释放旧 task 的话，旧 task 的栈还在跑 schedule 的内容，没有完全退栈，这样别的 cpu 拿到这个 task 的时候就无法运行了。

三，印象深刻的bug

正如第二节所说，由于我的 task 要在新 task 发生中断之后才被释放，这势必会导致一个 task 在某段时间处于没有 cpu 在运行的状态。为了缩短这个时间。我让 schedule 在抽取到自己当前运行的 task 时，无视 is_running 位（因为这个时候运行这个 task 的就是这个 cpu 自己啊）。但是我忘记检查 block 位了。所以有一种情况是信号量在获取资源失败的时候 block 了当前 task 然后 yield 进行上下文切换，然后 schedule 又抽到了自己的task，无视了所有标记位然后继续让 cpu 运行这个 task。由于这个 bug 实在 信号量测试里面暴露出来的，导致我浪费了好多时间检查信号量的实现，最后走投无路的时候在 schedule 里面加入了 panic_on 检查调度到的 task 的 block 位才发现是 schedule 的 bug。

这次实验完成的速度比 L1 快很多，大概是因为 L1 的 de 了一个月的 bug 的洗礼让我熟练的掌握了各种 debug 的技巧。比如看懂 makefile 并且修改 makefile，利用条件编译在命令行通过不同的命令运行不同的测试。比如使用 log 打印各个函数的进出情况，比如使用 panic_on 防御性检查各种标记录位的合法性，比如使用 gdb + vscode 远程调试，监视变量在运行过程的值。熟练掌握这些技巧之后，bug 也容易发现的多。现在最喜欢的 debug 模式是写测试暴露 bug，然后使用 panic_on，gdb，log 定位 bug 的位置。这样调试的效率要高很多。

L3实验报告

一，架构设计

主要是虚拟内存的管理，实现了 C-O-W，页的映射全部在页错误里面完成。使用了链表管理虚拟内存的 flag，引用计数记录在一个与物理页一一对应的数组里。虚拟页与物理页的映射也是用链表存储的。

二，实验感受

这个实验我没有完成中断嵌套的功能。即使这样代码量也来到了700行，作为一个实验来说缺确实比较多。实验里的一些不太让我舒服的设计主要有不开放页表，需要自己写数据结构存储 C-O-W 标记，存储引用计数。可读写权限暂且不提，因为在页缺失处理时确实需要知道 map 时的 protect 权限。但是前面两个应该都是可以打在页表里的。也就是说这个信息是已有的，但是为了统一不同架构的 Address Space 隐藏起来了。这在精简的内核设计里面感觉很浪费，很不优雅。事实上这种为了兼容性舍弃一部分信息的理由感觉也很站不住脚，因为涉及到 ABI，本身就没法多么兼容。除非使用条件编译，不过既然都使用条件编译了，还需要这种代码层次的兼容性干嘛呢，无非就是代码量多少而已。总而言之，感觉对这种不开放页表相关 api 的行为不是很理解。

事实上，我从 `am` 的 `vme.c` 里面找到了我想要的页表信息，以及查询方法。但是相关的函数是 static，并且也没有开放 api，或者提供帮助函数。我当然可以直接自己写一个读取页表的函数，毕竟我有 `AddrSpace` 但是这样感觉有违实验设计者的设计初衷，所以还是自己保存了相关的信息。

这次试验收获最大的就是理解了 C-O-W 的行为以及各种系统调用的行为。以前当黑盒用的一些命令现在都明白了其中的原理，对线程的管理也更加清楚了。