OS 第五次作业

于新雨 计25 2022010841

第一题

主要参考的是 kernel docs 和 linux 官方文档 的描述
具体来说，物理内存布局往往没那么简单，当然一个很直接的想法是从 0x0 地址往上开始一段连续的空间，但是可能有以下情况：

- 这段空间有若干我们不能访问的地方
- 是以若干段连续空间组成的
- 考虑多核，则有 [NUMA] 机制(https://en.wikipedia.org/wiki/Non-uniform_memory_access)

linux 抽象出来了 FLAT_MEM 和 SPARSE_MEM 两个不同的模型，但是他们的共同点都是：物理内存按页组织，物理栈帧以结构体的形式表示，存在数组之中，且每个结构体和对应的物理栈帧有一对一的映射关系

具体内存的布局方法参考 这里

具体来说，linux 下面有三个 memory zones, 分别如下

- ZONE_DMA (支持 direct memory mapping，给某些硬件使用的，是在< 16MB 的位置)
- ZONE_NORMAL (正常被 mapped 和使用的物理页，在 16 - 896MB)
- ZONE_HIGHMEM (动态 mapped 在大于 896 MB 地址的位置)
  且上述地址区间是对于 linux x86 下来说的

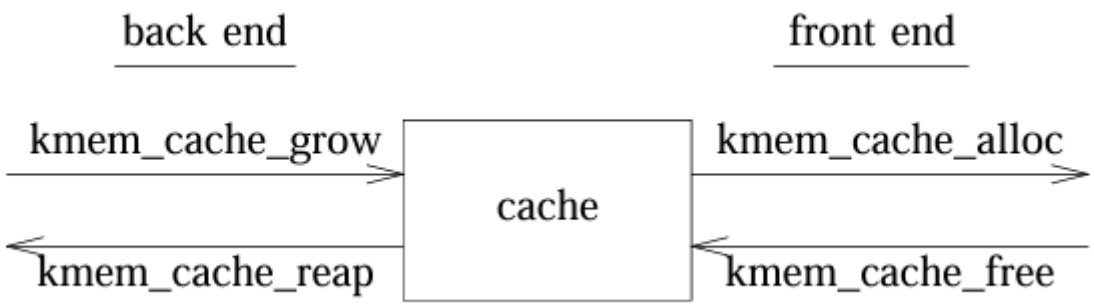
第二题

首先 slab allocator 现在基本不再使用，见 ctf-wiki 的介绍，现在主要是 slub allocator
较多的利用了程序的局部性，实现了缓存等机制
cache 对分配和释放对象的思路如下：

1. 分配对象
   - 如果在 cache 可以找到现有的，就直接拿来用
   - 否则分配内存给它，新建一个对象
2. 释放对象
   - 直接放到 cache 里面并且不调用 destructor
3. 回收内存
   - 析构对象，然后把对应内存回收了

实现

cache



frontend 和 allocator 相连，完成上述分配和释放对象功能，backend 则是管内存，比如说是内存的申请和回收

slabs

是 slab allocator 管理的单位，allocator 申请和释放内存都是以整个 slab 的构造和回收为单位

一个 slab 包含若干虚拟地址连续的页，分成等大的 chunk 然后维护多少个 chunk 被分配了

这个实现很简洁，但是优势很大，比如回收内存简单（只要当前 slab 所有的 chunk 都收回了就能回收了），分配和释放内存很快，且不会有啥 fragmentation

对于小的内存申请，slab 把一个页分成若干等大的部分，申请的时候给一个给当前对象

大的内存申请的话，就是物理的 memory layout 和逻辑的 memory layout 相同了（个人理解是 slab allocator 不会往头/尾插控制数据）

free list

双层 freelist 如下：

1. 每个 cache 有管理所有 slab 的双向环形链表
2. 每个 slab 对于自己管理的 buffer 也维护一个 free list

slab coloring

对于 cache 的管理，需要之前计组的知识，对于多路组相联的 cache，或者直接映射的 cache，都是以地址的低若干位算出该放到哪个 cache 里面

这里在 slab 的做法就是，之前说过对于小的内存申请，是在同一个页里面做划分然后申请的时候依次给出去，则它划分的第一个 buffer 相对于页基址的地址，对于每个 slab 是不同的且相差一个小距离，就可以回避到把一堆 free 的对象塞到同一个 cache 行里面造成的冲突