Jos Lab2 Design Report

5140219396
陆伟嘉

Part A:

Boot_alloc():
这个函数做两件事，返回当前 free 对应地址，同时将 free 移至对应位置以避免刚刚申请的空间被重复分配。由于要 page 对齐，所以只需在当前地址加上 n 后取下 ROUNDUP 即可。
Mem_init():
第一部分这个函数只需要申请 npages 个 struct page 物理页结构大小的空间来存放这些 struct 即可，直接调用刚写好的 boot_alloc()即可。
Page_init():
由于有一些物理页已经被用作映射其他用处，不能再被申请分配，所以初始化时不能将他们加入 page_free_list。这三部分分别是，第 0 页、IOPHYSMEM->EXTPHYSMEM、EXTPHYSMEM 上分配的 kernel 自身数据部分、页目录（PGSIZE）和 Page 结构体部分。
Page_alloc():
Page_free():
对自己 alloc 出来的物理页进行调度管理，简单的链表操作，不再赘述。

Part B:

Pgdir_walk():
这个函数是一个页表的查找函数，返回 pte 的指针。这个函数的逻辑在注释中已经很明显了，关键在于函数中涉及到的几个地址应该是物理地址还是虚拟地址让我纠结了很久。首先要填页目录中的项，这要填页表的首地址，应该填物理地址。究其原因，因为地址的翻译就需要参考页表，若页表中填的仍是虚拟地址，寻址过程就会陷入无尽循环。另外一个纠结的地址就是返回地址，这个地址却应该是虚拟地址。原因在于，对于进程，所有的操作都是基于虚拟地址空间，物理地址丢给一个进程并没有实际作用，因为所有访存都绕不开页表，所以只有页表中填的是物理地址，其他地址应该都是虚拟地址。
boot_map_region()：
一个循环，其实说白了就是一个填页表的过程。
page_lookup()：
其实是一个加强版的 pgdir_walk()，通过 pte_store 可以达到 pgdir_walk()功能，而返回值返回了对应的物理页的 page 结构体。如果说 pgdir_walk()是页表查找（找到那一项），page_lookup()则是页的翻译（找到对应的物理页）。
page_remove()：
释放虚拟地址的页，找到 pte 和 page 结构体，分别进行释放（pte 清零，page-ref 减一）

page_insert():
我感觉和 page_map_region()大同小异，硬要说什么不同的话，或许 map 只是映射填页表，而 insert 则是调用后的填页表，本质上就是 ref 加不加的区别。Insert 函数有一个 tricky 的地方在于重复映射，为了防止该物理页被释放，只需要将 ref 加一的操作放在 remove 之前即可，这样就不用针对这种情况写特例了。

Part C:

Boot_map_region_large():
本质上很 boot_map_region 一样，只是针对大页表，所以不用访问二级页表，且每次映射一个 PTSIZE，flag 多设一个 PTE_PS。
Mem_init():
将之前写的“业务层”代码真正落实的部分，将要求的内存区域一一映射，以供 os 管理。就是测试中遇到了两个问题，先是要将 kernel 部分的映射切换为大页表模式，第二个就是切换后 qemu 不断重启，原因在于我忘记设置 cr4 了，仿照 cr0 的操作写一下就好了。

总结：

整个内存管理分为三层，page_alloc(),page_free()这类是最底层，直接对物理页进行管理。而 page_insert(),page_remove()等相当于“业务层”，进行地址映射，页表修改，page 结构体 ref 管理等。而 mem_init()位于最高层，是整个 kernel 部分虚拟内存管理的逻辑函数，相当于“表现层”（好像比方有些不准确~~）。而各个函数调用的“服务函数”应该就是其下的一层，而不应该跨层调用造成混乱。