Homework #2

周雨扬
2000013061

1 Challanges

本次作业完成了所有的代码补全任务，做了如下的 Challange:

- Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter `showmappings 0x3000 0x5000` to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses `0x3000`, `0x4000`, and `0x5000`.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.

2 Exercise 1

boot_alloc()

该部分代码需要初始的时候设置用于声明一级页表，以及管理二级页表的链表的内存 (事实上没有声明二级页表)，等待两者初始化完毕后再利用这个没有建立完的二级页表进行寻址。由于初始没有二级页表，这部分即用于声明二级页表，以及页管理链表的内存。

虽然代码中，声明的内存只有大约 `4 MB`，远不会超出内存限制；但是安全起见还是加上了防止内存溢出的设置。代码可以见 `kern/pmap.c, Line 106 ~ 117`。

mem_init()

此时我们只需要声明用于存储页链表 `pages` 的内存即可。因此可以通过直接调用 `boot_alloc` 解决。代码可以见 `kern/pmap.c, Line 163 ~ 165`。

page_init()

这里我们需要初始化链表。观察链表元素我们发现 `pp_ref` 是表示页被多少玩意引用的，如果是 0 则代表是一个空页；同时 `pp_link` 记录了空页链表的下一个页的链接，如果其不是空页则指针为空。

根据信息知道此时候只有三部分已经使用的页: 第一部分是一级页表，共 1 页；第二部分是 IO 缓冲区，界限已经给出；第三部分是我们声明链表的内存 (有可能还包括其他的)，界限可以通过调用 `boot_alloc()` 获得。

据此我们可以根据界限分类声明即可，代码可以见 `kern/pmap.c, Line 269 ~ 292`。

page_alloc()

这里我们需要支持页分配；此时我们只需要从 page_init() 预处理出来的 page_free_list 中寻找即可，注意链表维护的细节即可。代码可以见 kern/pmap.c, Line 312 ∼ 320。

page_free()

这里我们需要支持页收回。我们需要通过检查 pp_ref 是否为 0 来判断是否仍有页引用；需要通过检查 pp_link 判断该页是否已经在空列表中。如果满足条件，则直接将其加入列表即可。代码可以见 kern/pmap.c, Line 333 ∼ 338。

3 Exercise 4

pgdir_walk()

这部分需要我们检索二级页表，如果其不存在且需要创建则声明对应的页。

首先我们检索一级页表，如果一级页表中 PTE_P 位为真，则表示其对应的二级页表存在，直接根据访存地址即可确定存储页信息的位置；如果一级页表不存在，且我们需要创建，则我们首先需要声明一页用于存储二级页表，修改一级页表的信息，修改页引用数，最后返回存储页信息的地址。

注意返回地址时候的，对地址进行加法操作的时候指针类型。代码可以见 kern/pmap.c, Line 378 ∼ 394。

boot_map_region()

这部分需要将一段连续虚拟地址映射到连续物理地址上，同时设置权限位信息。

首先利用 pgdir_walk(), 我们找到所虚拟地址对应的所有的页，如果不存在上述的页还需要将其创建。之后根据返回的地址，我们只需要修改其映射的物理地址，以及其对应的权限位即可。

代码可以见 kern/pmap.c, Line 412 ∼ 417。

page_lookup()

这部分需要将在二级页表中查询虚拟地址对应的页信息，以及对应的二级页表的地址。

利用 pgdir_walk() 查询的结果，如果存在于页表中，且 PTE_P 位为真，则说明其存在，返回对应信息即可；否则返回不存在。代码可以见 kern/pmap.c, Line 476 ∼ 486。

page_remove()

这部分需要删除虚拟地对应的页。

注意在删除的时候，虽然可能页引用数量仍不为 0，其仍需要在页表中删除。代码可以见 kern/pmap.c, Line 508 ∼ 515。

page_insert()

这部分需要加入虚拟地对应的页。

此时我们不仅需要查询对应的页，还需要在该页已经存在的时候删除原有的页映射信息；删除后既可以通过修改对应映射信息来插入。

注意，我们不仅需要加入二级页表的权限信息，还需要将对应的修改一级页表的权限。代码可以见 `kern/pmap.c, Line 448 ~ 457`。

4 Exercise 5

mem_init()

此时我们可以直接使用 `boot_map_region` 进行虚拟内存到物理内存的映射。这里我们注意到我们没有必要进行栈缓冲区的映射，因为其不映射的话在寻址的时候就会报错，确实能够完成缓冲的问题。

代码可以见 `kern/pmap.c, Line 189,201,211`。

5 Question

Question 1

`uintptr_t`，因为 `T*` 的类型是 `uintptr_t`

Question 2

`Block 961~1024, Address 0xf0000000~0xffffffff` 最顶上的 64 个一级页表，将虚拟内存映射到了 `256MB` 的物理内存中。

`Block 960, Address 0xefc00000 ~ 0xeffffffff`, 映射了内核栈，内核信息等所处的内存。

Question 3

第 1 段一级页表映射了包括一级页表，二级页表，页链表等的信息的，

由于二级页表中每一个存储信息的结构 `pte_t` 中有 12 位的专门用于存储权限的内存空间，这些比特表示用户/系统是否有权限来对这页的信息进行相关操作。

在 `mem_init` 中我们一开始已经设定了存储内核信息，内核栈的部分的权限，我们将其设置为不可阅读。因此在用户权限的进程尝试访问的时候，系统会终止访问并返回错误。

Question 4

该内核共有 1 个一级页表，1024 个二级页表，至多 2^20 个大小为 $4KB$ 的页，因此能够管理的内存至多有 $4GB$。

Question 5

初始其会声明至多 $8MB$ 用于存储页链表信息，$4MB$ 用于二级页表，$4MB$ 用于一级页表。

Question 6

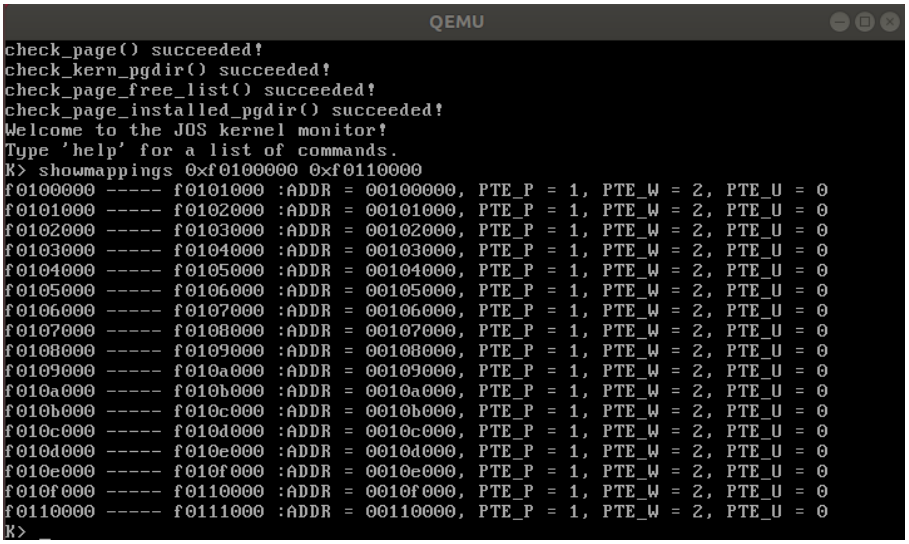
代码片段位于 `kern/entry.S, Line 67 ~ 69`

没有设置的时候内存映射中虚拟内存 $[0, 4MB]$ 被映射到了物理内存 $[0, 4MB]$，因此以一个较低的 `EIP` 运行不会产生问题，但是在设置后虚拟内存 $[0, 4MB]$ 便不再允许随意调用了。

6 Challange Implementation

challange 1

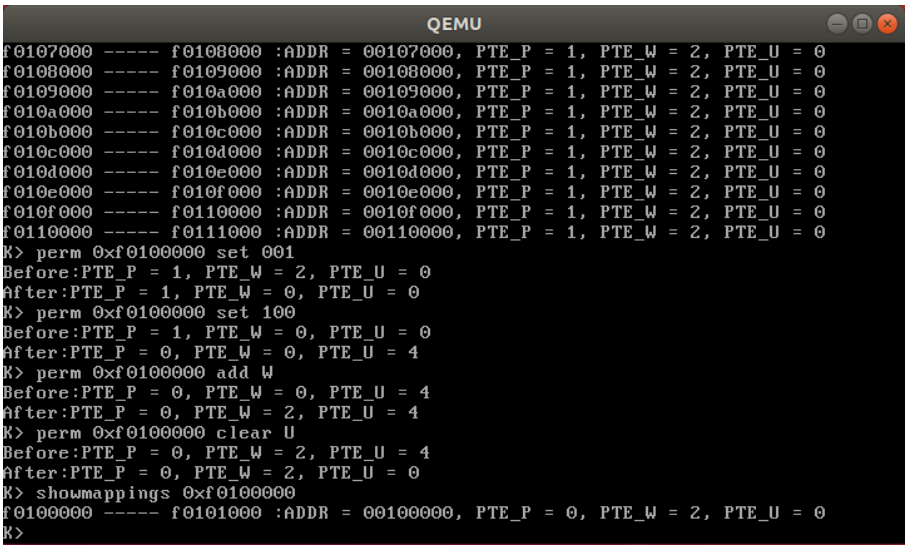
第一个 challange 我们需要逐页的输出某一段连续虚拟地址对应的物理地址位置信息，访问权限，以及在不存在上述映射时候返回错误信息。最终实现效果如下：



代码可以见 `kern/moniter.c, Line 95 ~ 129`。

challange 2

第二个 challange 我们需要修改某一个页表的权限位，最终实现效果如下：



代码可以见 `kern/moniter.c, Line 133 ~ 175`。