困难一：对于 get_pte 的返回值不太理解

```
return &((pte_t*)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
```

KADDR(PDE_ADDR(*pdep)):这部分是由页目录项地址得到关联的页表物理地址， 再转成虚拟地址

PTX(la)：返回虚拟地址 la 的页表项索引

最后返回的是虚拟地址 la 对应的页表项入口地址

我认为一个虚拟地址根据二级页表找到页表项的地址，这个地址应该是个物理地址，但是在这个位置返回的是一个页表项对应的虚拟地址入口。不明白为什么不直接返回物理地址而返回虚拟地址。

后来得知原因，是因为在程序中，一般使用的是逻辑地址或虚拟地址，而且也比较方便和容易使用。

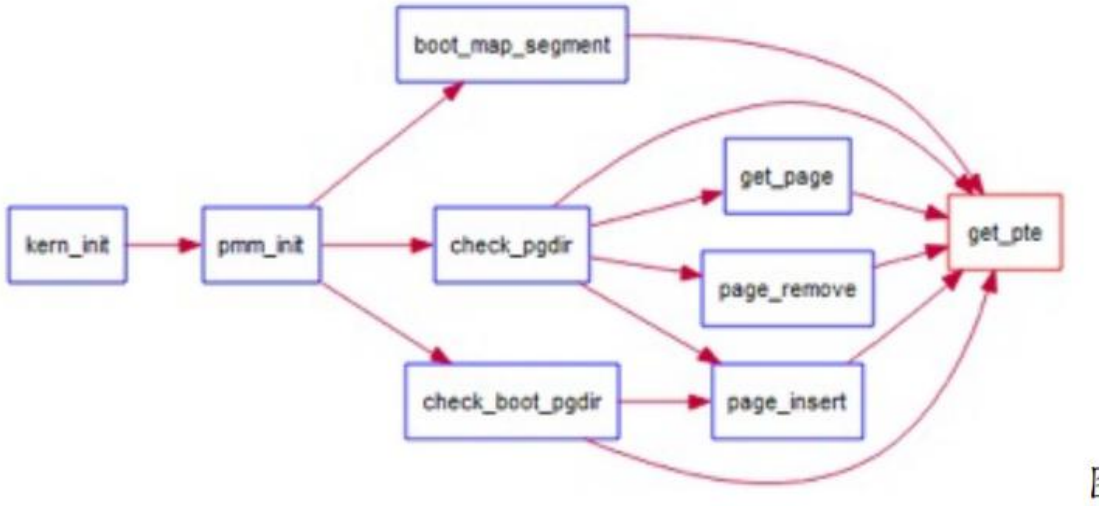
困难二：实验中完成的只有琐碎的代码编写工作，只了解单个函数或几个函数的功能，对于整个实验的整体执行流程不太清除，也就是调用关系太复杂、不明确，而对于一个函数一个函数的阅读和分析工作又太繁杂，工作量太大。

对于这个问题，一开始我就是直接在函数里面看调用了哪个函数然后就跳转到那个函数里面去分析，但是函数里面一层接一层的调用，使我对这些调用关系就更加混乱与不理解了。解决的办法就是找到一个代码分析的工具，可以很好的弄清大项目的函数调用关系，这里的软件是 understand。

在实验中，明确调用关系图后，就可以只了解调用函数的功能，

不需要去了解调用函数里面的具体实现，这对代码的可读性和可理解性大大提高，也对项目的整体逻辑流程更快更方便的理解。

例：



get_pte 的函数调用关系图

困难三：对页目录表项（PDE）和页表项（PTE）的具体结构以及区别不清楚，不明白。

页目录表项（PDE）

- 前 20 位表示 4K 对齐的该 PDE 对应的页表起始位置（物理地址，该物理地址的高 20 位即 PDE 中的高 20 位，低 12 位为 0）；
- 第 9-11 位未被 CPU 使用，可保留给 OS 使用；
- 接下来的第 8 位可忽略；
- 第 7 位用于设置 Page 大小，0 表示 4KB；
- 第 6 位恒为 0；
- 第 5 位用于表示该页是否被使用过；

- 第 4 位设置为 1 则表示不对该页进行缓存；
- 第 3 位设置是否使用 write through 缓存写策略；
- 第 2 位表示该页的访问需要的特权级；
- 第 1 位表示是否允许读写；
- 第 0 位为该 PDE 的存在位；

页表项（PTE）

- 高 20 位与 PDE 相似的，用于表示该 PTE 指向的物理页的物理地址；
- 9-11 位保留给 OS 使用；
- 7-8 位恒为 0；
- 第 6 位表示该页是否为 dirty，即是否需要在 swap out 的时候写回外存；
- 第 5 位表示是否被访问；
- 3-4 位恒为 0；
- 0-2 位分别表示存在位、是否允许读写、访问该页需要的特权级；

区别就是页表项结构有一个修改位，表明页表项对应的物理页是否修改过。

吐槽一：代码太多，阅读量太大。

吐槽二：调用关系比较复杂，实验对应的部分从整体上看难以理解。