lab3

练习1:给未被映射的质地映射上物理页

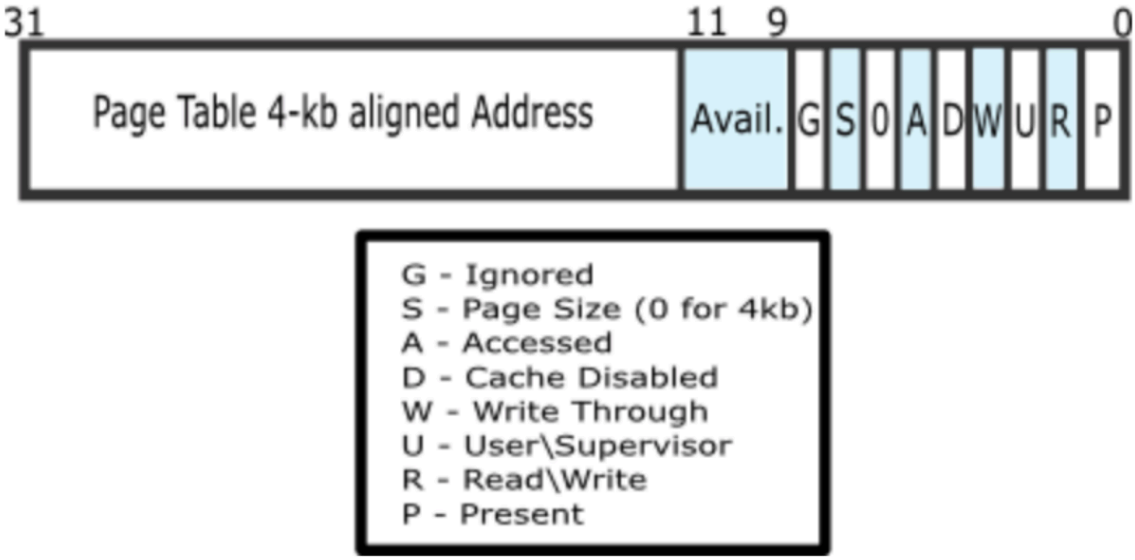
首先与其他中端的处理相似的，硬件会将程序状态字压入中端栈中，与ucore中的部分中端处理代码一起建立起一个trapframe，并且硬件还会将出现了异常的线性地址保存在cr2寄存器中；
与通常的中断处理一样，最终page fault的处理也会来到trap_dispatch函数，在该函数中会根据中断号，将page fault的处理交给pgfault_handler函数，进一步交给do_pgfault函数进行处理，因此do_pgfault函数便是我们最终需要用来对page fault进行处理的地方；
接下来对do_pgfault函数进行分析：

1.该函数的传入参数总共有三个，其中第一个是一个mm_struct变量，其中保存了所使用的PDT，合法的虚拟地址空间（使用链表组织），以及与后文的swap机制相关的数据；而第二个参数是产生pagefault的时候硬件产生的error code，可以用于帮助判断发生page fault的原因，而最后一个参数则是出现page fault的线性地址（保存在cr2寄存器中的线性地址）；
2.在函数中，首先查询mm_struct中的合法的虚拟地址(事实上是线性地址，但是由于在ucore中弱化了段机制，段仅仅起到对等映射的作用，因此虚拟地址等于线性地址)链表，用于确定当前出现page fault的线性地址是否合法，如果合法则继续执行调出物理页，否则直接返回；
3.接下来使用error code（其中包含了这次内存访问是否为读/写，以及对应的物理页是否存在），以及查找到的该线性地址的内存页是否允许读写来判断是否出现了读/写不允许读/写的页这种情况，如果出现了上述情况，则应该直接返回，否则继续执行page fault的处理流程；
4.接下来根据合法虚拟地址（mm_struct中保存的合法虚拟地址链表中可查询到）的标志，来生成对应产生的物理页的权限；
5.接下来的部分则是在本练习中需要完成代码补全的部分，首先使用在lab2中实现的函数get_pte来获取出错的线性地址对应的虚拟页起始地址对应到的页表项，在ucore中同时使用页表项来保存物理地址（在Present位为1的时候）以及被换出的物理页在swap外存中的位置（以页为单位，每页大小刚好为8个扇区，此时P位为0），并且规定swap中的第0个页空出来不用于交换，因此如果查询到的PTE不为0，则表示对应的物理页可能在内存中或者在外存中（根据P位决定），否则则表示对应的物理页尚未被分配，此时则需要调用在lab2中实现的内存分配功能来获取对应的物理页，并且将其与当前的虚拟页设置上映射关系，这个部分在lab3中被封装成了pgdir_alloc_page函数；根据上述分析练习1中需要编写的代码实现如下：

```
ptep = get_pte(mm->pgdir, addr, 1); // 获取当前发生缺页的虚拟页对应的PTE
if (*ptep == 0) { // 如果需要的物理页是没有分配而不是被换出到外存中
    struct Page* page = pgdir_alloc_page(mm->pgdir, addr, perm); // 分配物理页，并且与对应的虚拟页建立映射关系
} else {
    // 将物理页从外存换到内存中，练习2中需要实现的内容
}
```

问题1：页目录项和页表项对页替换算法的用处

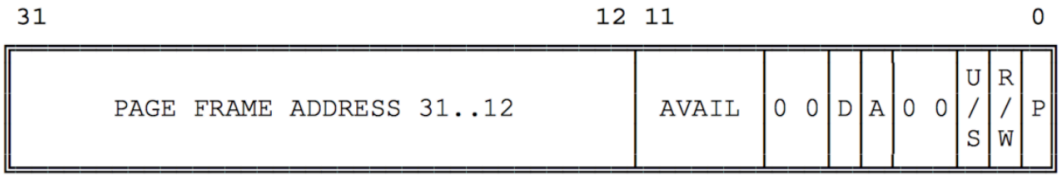
页替换涉及到换入换出，换入时需要将某个虚拟地址vaddr对应于磁盘的一页内容读入到内存中，换出时需要将某个虚拟页的内容写到磁盘中的某个位置。而页表项可以记录该虚拟页在磁盘中的位置，为换入换出提供磁盘位置信息。页目录项则是用来索引对应的页表。



先描述页目录项的每个组成部分，PDE（页目录项）的具体组成如下图所示；描述每一个组成部分的含义如下[1]：

- 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；




```
P       - PRESENT
R/W     - READ/WRITE
U/S     - USER/SUPERVISOR
D       - DIRTY
AVAIL   - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

接下来描述页表项（PTE）中的每个组成部分的含义，具体组成如下图所示[2]：

- 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；
- 通过上述分析可以发现，无论是页目录项还是页表项，表项中均保留了3位供操作系统进行使用，可以为实现一些页替换算法的时候提供支持，并且事实上在PTE的Present位为0的时候，CPU将不会使用PTE上的内容，这就使得当P位为0的时候，可以使用PTE上的其他位用于保存操作系统需要的信息，事实上ucore也正是利用这些位来保存页替换算法里被换出的物理页的在交换分区中的位置；此外PTE中还有dirty位，用于表示当前的页是否经过修改，这就使得OS可以使用这个位来判断是否可以省去某些已经在外存中存在着，内存中的数据与外存相一致的物理页面换出到外存这种多余的操作；而PTE和PDE中均有表示是否被使用过的位，这就使得OS可以粗略地得知当前的页面是否具有着较大的被访问概率，使得OS可以利用程序的局部性原理来对也替换算法进行优化(时钟替换算法中使用)；

问题2：缺页服务例程出现页访问异常时，硬件需要做哪些事情

1. 关闭中断。
2. 线性地址存储在CR2中，并且把表示页访问异常类型的值（简称页访问异常错误码，errorCode）保存在中断栈中。CPU在当前内核栈保存当前被打断的程序现场，即一次雅茹当前被打断程序使用的FEFLAGS，CS，EIP，errorCode；由于页的访问中号0xE对用的中断服务例程的地址加载到CS和EIP寄存器中。ucore开始执行中断服务例程，保存硬件没有保存的寄存器，中断号、DS、ES和其他通用寄存器都压入栈。现场保护完毕，trap.c的trap函数开始了中断服务例程的处理。

练习2：补充完成基于FIFO的页面替换算法

根据练习1中的分析，可以知道最终page fault的处理会转交给do_pgfault函数进行处理，并且在该函数最后，如果通过page table entry获知对应的物理页被换出在外存中，则需要将其换入内存，之后中断返回之后便可以进行正常的访问处理。这里便涉及到了以下若干个关于换入的问题：
应当在何处获取物理页在外存中的位置?
物理页在外存中的位置保存在了PTE中；
如果当前没有了空闲的内存页，应当将哪一个物理页换出到外存中去?
这里涉及到了将物理页换出的问题，对于不同的算法会有不同的实现，在本练习中所实现的FIFO算法则选择将在内存中驻留时间最长的物理页换出；
在进行了上述分析之后，便可以进行具体的实现了，实现的流程如下：
判断当前是否对交换机制进行了正确的初始化；
将虚拟页对应的物理页从外存中换入内存；
给换入的物理页和虚拟页建立映射关系；
将换入的物理页设置为允许被换出；

```
ptep = get_pte(mm->pgdir, addr, 1); // 获取当前发生缺页的虚拟页对应的PTE
if (*ptep == 0) { // 如果需要的物理页是没有分配而不是被换出到外存中
    struct Page* page = pgdir_alloc_page(mm->pgdir, addr, perm); // 分配物理页，并且
与对应的虚拟页建立映射关系
} else {
        if (swap_init_ok) { // 判断是否当前交换机制正确被初始化
            struct Page *page = NULL;
            swap_in(mm, addr, &page); // 将物理页换入到内存中
            page_insert(mm->pgdir, page, addr, perm); // 将物理页与虚拟页建立映射关系
            swap_map_swappable(mm, addr, page, 1); // 设置当前的物理页为可交换的
            page->pra_vaddr = addr; // 同时在物理页中维护其对应到的虚拟页的信息，这个
```

```
语句本人觉得最好应当放置在page_insert函数中进行维护，在该建立映射关系的函数外
对物理page对应的虚拟地址进行维护显得有些不太合适
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
}
```

对上述代码进行分析，发现当调用swap_in函数的时候，会进一步调用alloc_page函数进行分配物理页，一旦没有足够的物理页，则会使用swap_out函数将当前物理空间的某一页换出到外存，该函数会进一步调用sm（swap manager）中封装的swap_out_victim函数来选择需要换出的物理页，该函数是一个函数指针进行调用的，具体对应到了_fifo_swap_out_victim函数（因为在本练习中使用了FIFO替换算法），在FIFO算法中，按照物理页面换入到内存中的顺序建立了一个链表，因此链表头处便指向了最早进入的物理页面，也就在在本算法中需要被换出的页面，因此只需要将链表头的物理页面取出，然后删掉对应的链表项即可；具体的代码实现如下所示：

```
list_entry_t *head=(list_entry_t*) mm->sm_priv; // 找到链表的入口
assert(head != NULL); // 进行一系列检查
assert(in_tick==0);
list_entry_t *le = list_next(head); // 取出链表头，即最早进入的物理页面
assert(le != head); // 确保链表非空
struct Page *page = le2page(le, pra_page_link); // 找到对应的物理页面的Page结构
list_del(le); // 从链表上删除取出的即将被换出的物理页面
*ptr_page = page;
```

在进行page fault处理中还有另外一个与交换相关的函数，swap_map_swappable，用于将指定的物理页面设置为可被换出，分析代码可以发现，该函数是sm中的swap_map_swappable函数的一个简单封装，对应到FIFO算法中实现的_fifo_swap_map_swappable函数，这也是在本次练习中需要进行实现的另外一个函数，这个函数的功能比较简单，就是将当前的物理页面插入到FIFO算法中维护的可被交换出去的物理页面链表中的末尾，从而保证该链表中越接近链表头的物理页面在内存中的驻留时间越长；该函数的一个具体实现如下所示：

```
list_entry_t *head=(list_entry_t*) mm->sm_priv; // 找到链表入口
list_entry_t *entry=&(page->pra_page_link); // 找到当前物理页用于组织成链表的
```

```
list_entry_t
assert(entry != NULL && head != NULL);
list_add_before(head, entry); // 将当前指定的物理页插入到链表的末尾
```

如果要在ucore上实现"extended clock页替换算法"，请给出你的设计方案，现有的swap_manager框架是否足以支持在ucore中实现此算法？如果是，请给出你的设计方案。如果不是，请给出你的新的扩展和基此扩展的设计方案。并需要回答如下问题：

需要被换出的页的特征是什么？

原则上只有映射到用户空间的且被用户程序直接访问的页面才能被交换出，而被内核直接使用的内核空间的页面不能被交换出去。但是在此实验中还没有设计用户态执行的程序，所以仅通过执行check_swap函数在内核中分配一些页，模拟对这些页的访问，然后通过swap_map_swappable函数来查询这些页的访问情况并间接调用相关函数，换出不常用的页到磁盘上。