# Lab2

## 变量和结构体

### 内存组织图

openeuler的内核中关于内存的组织是基于节点和页的。

组织结构如下：

zone1

|  |  |
| --- | --- |
| 起始页帧 | zone\_start\_pfn |
| 所属pglist | \*zone\_pgdat |
| 空闲区 | free\_area[MAX\_ORDER] |
| 页统计 |  |
| ... |  |

pglist\_data

|  |  |
| --- | --- |
| 页分区存储 | zone1  zone2... |
| zonelist |
| LRU列表 |  |
| ... |  |

node1

node2

...

noden

lruvec

|  |  |
| --- | --- |
| 5种lru列表 | lists[NR\_LRU\_LISTS] |
| 所属pglist |  |
| ... |  |

5种lru lists

|  |
| --- |
| LRU\_INACTIVE\_ANON |
| LRU\_ACTIVE\_ANON |
| LRU\_INACTIVE\_FILE |
| LRU\_ACTIVE\_FILE |
| LRU\_UNEVICTABLE |

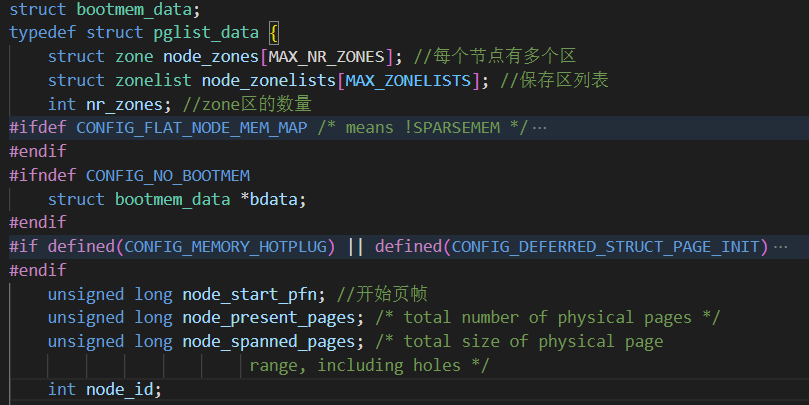
|  |
| --- |
| 页1 |
| 页2 |
| ... |
| 页n |

1. 在NUMA机器上每个NUMAnode都有一个pglist\_data结构体；而UMA只有一个pglist\_data。
2. 每个pglist\_data节点被划分为多个zone区
3. 每个区中有若干页
4. 每个pglist\_data中有个lru\_vec结构体，管理本节点内的lru列表。lru列表中有5种lru链表，分别记录着不同类型的页
5. 对于一个页，如果没有分配且是连续空闲区的第一个页则加入到free(空闲)列表；如果分配了加入到相应的lru列表；如果是slab的第一个页加入到相应的链表

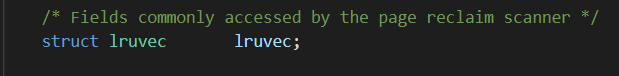
### 相关结构体

#### 2.1 pglist\_data 节点

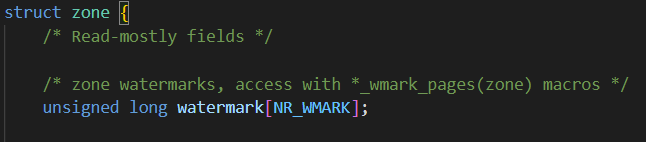
内存的最高抽象，节点。



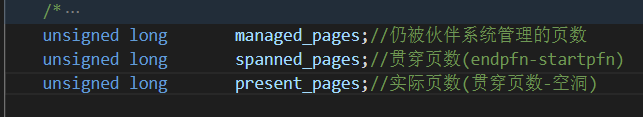
包含一个lruvec



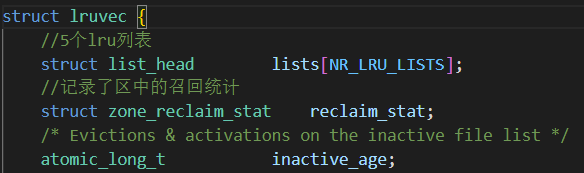
#### 2.2 zone节点划分的区。

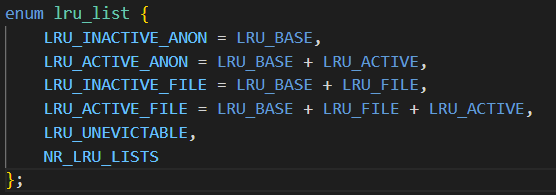




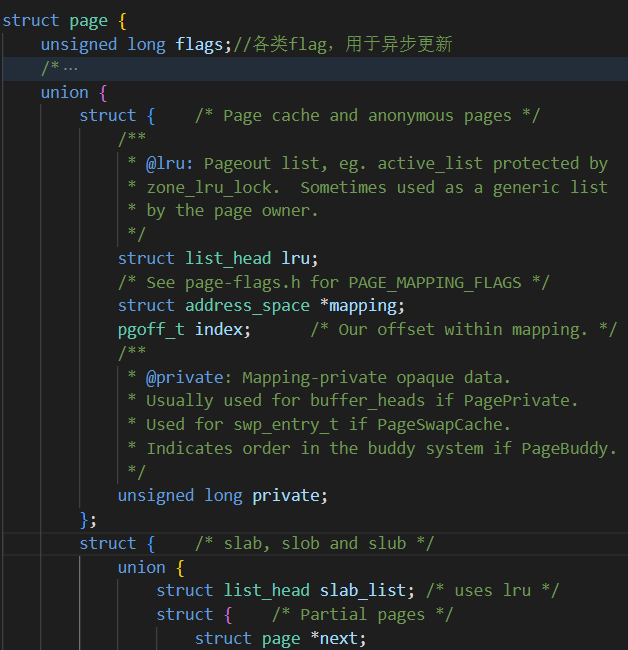


#### 2.3 lruvec lru主要结构体 定义了5种lru链表





#### 2.4 page 每个page的唯一标识和记录



1. 由于linux中内存管理的机制非常复杂，页有很多不同的类型，所记录的内容自然不同，所以用了union来复用空间。
2. unsigned long flags 记录了页的各类状态，有时会用于异步更新状态

其中和lru算法有关的有：

- PG\_active : 页是否活跃，活跃=1。

- PG\_referenced: 页是否被引用，每次被引用set为1。

- PG\_lru: 页是否在lru链表中。

- PG\_mlocked: 页被锁在内存中，不能换出和释放。如果加入lru链表则加入不可回收链表。

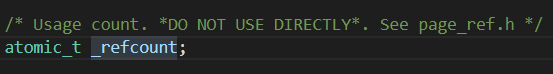
- PG\_swapbacked: 页依赖swap，如果=1，有三种可能：进程的匿名页（堆栈和数据段）；匿名mmap共享内存映射；shmem共享内存映射。用于swap判断。

1. struct list\_head lru ;

strcut list\_head slab\_list ;

页面根据不同情况会加入不同链表。例如一个页被进程使用时会加入对应的lru链表或者lru缓存；如果是slab的第一个页会加入slab链表中。

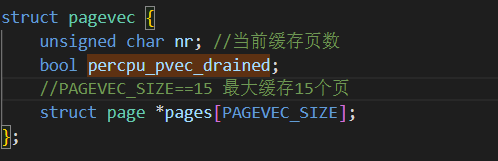
1. 用来记录页面被引用数



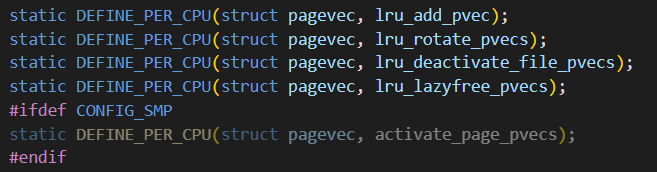
#### 2.5 pagevec 页缓存结构体

作用：由于每个节点(pglist\_data)中只有一个lruvec来控制所有lru链表；而一个节点有很多页，每次对lru链表操作都可能涉及锁操作，为了避免大量加锁解锁操作导致冲突，引入pagevec减少锁冲突。

1. pagecev 用于缓存最大15个页



1. 每个cpu的pagevec缓存有四/五种（缓存了4/5种lru链表的操作）



作用依次是：

- 加入lru链表

- 不活跃的头页放到尾部

- 把活跃文件页 放到不活跃文件链表

- 把活跃匿名页 放到不活跃匿名页链表

- 不活跃页放到活跃页（仅SMP架构）

## 函数

这部分只介绍主要的函数功能，具体功能实现在截图中用中文注释。函数调用和总体的页面淘汰流程在下一部分。

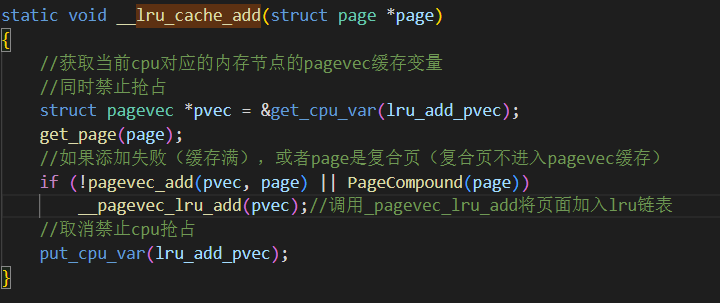
### lru函数

#### 1.1 lru\_cache\_add

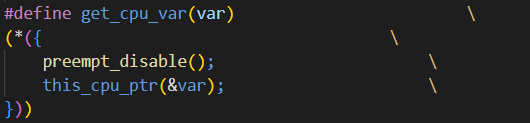
将输入的page添加到pagevec缓存。

阅读源码过程中发现关于lru\_cache\_add有好几个类似函数，但功能都是将页面加入pagevec缓存，应该是用于不同场景的，例如lru\_cache\_add\_file, lru\_cache\_add\_active\_or\_unevictable。

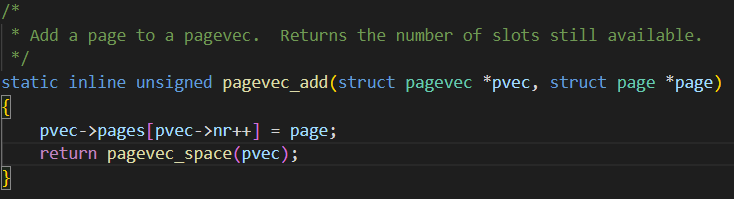
而且最后总会调用真正执行的函数：\_\_lru\_cache\_add



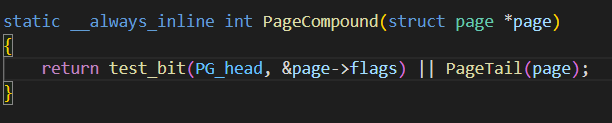
- get\_cpu\_var(var) 宏展开先禁止cpu抢占，返回当前处理器变量副本的地址。



- pagevec\_add(pvec, page)：将page加入到lru\_add\_pvec缓存，并且返回剩余缓存量。



- PageCompound(page)：判断page是不是复合页。



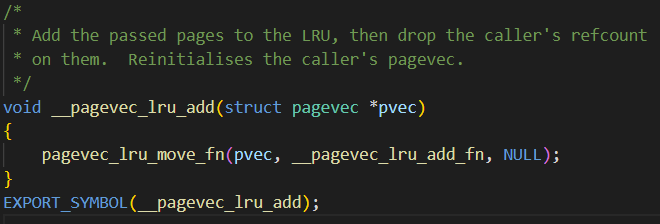
如果加入之后返回剩余缓存==0说明缓存已满；或者检测到是复合页则不加入pagevec，那么将调用\_\_pagevec\_lru\_add(pvec)把缓存中的页面加入到lru链表中，并且清空缓存。

#### 1.2 \_\_pagevec\_lru\_add

操作指示函数。

调用pagevec\_lru\_move\_fn函数并且以\_\_pagevec\_lru\_add\_fn为执行函数。

将lru\_add\_pvec缓存中的页面都加入到lru链表中并清空缓存。

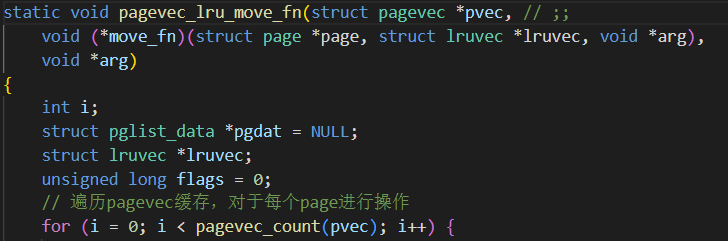


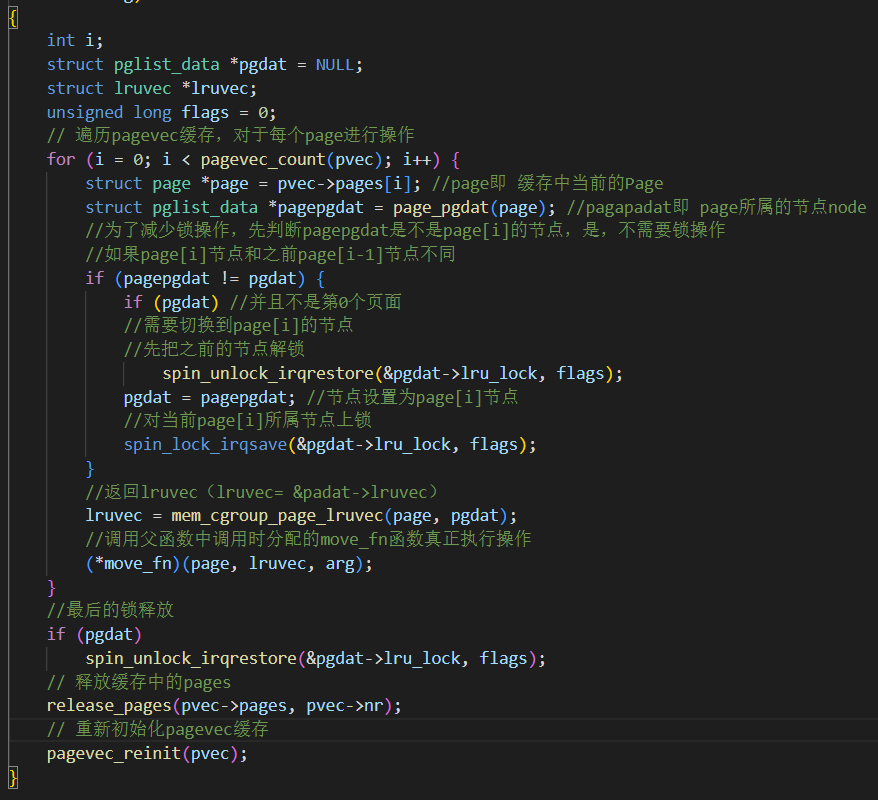
#### 1.3 pagevec\_lru\_move\_fn

操作执行函数。

会被不同的lru缓存指示函数调用，来完成指示的操作。

其中(\*move\_fn)(page, lruvec, arg)钩子函数用以赋值为各类操作的函数指针，真正实现lru链表操作。



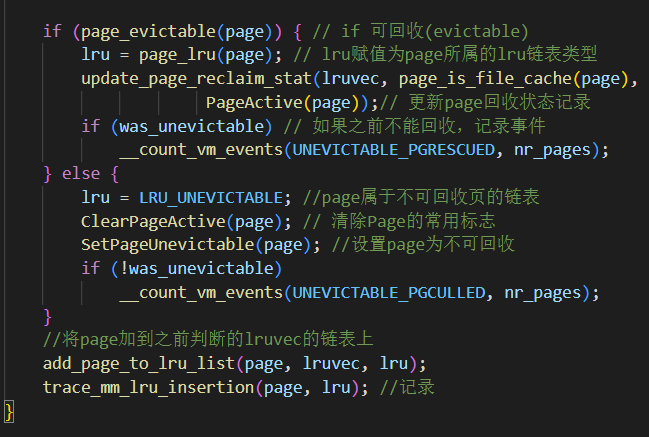


#### 1.4 \_\_pagevec\_lru\_add\_fn

操作执行函数。

将参数中的页加入相应的lru链表。

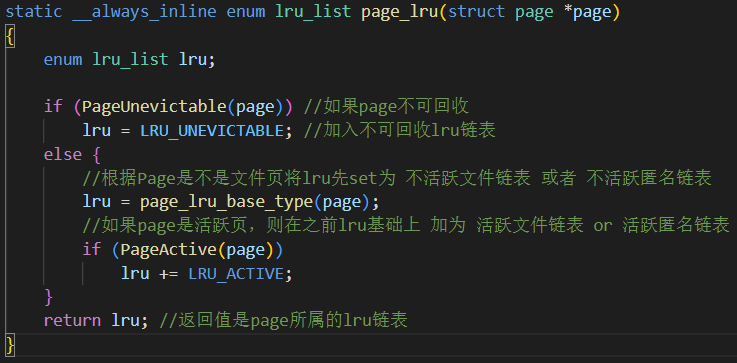




其中涉及到几个重要的功能函数，如下：

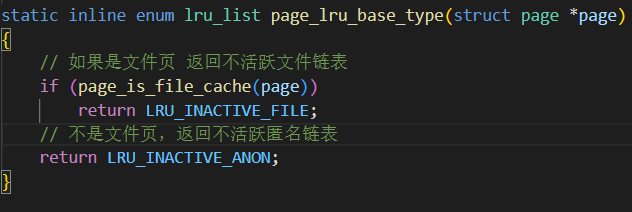
- page\_lru(page)：

判断page所属的lru链表类型并且返回该类型变量。

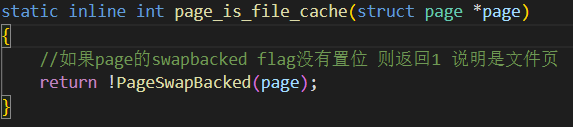


- page\_lru\_base\_type(page)：

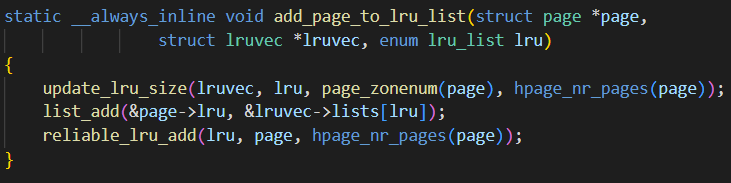
返回page属于文件or匿名不活跃链表



- page\_is\_file\_cache(page)：如果page是文件页返回1，否则返回0



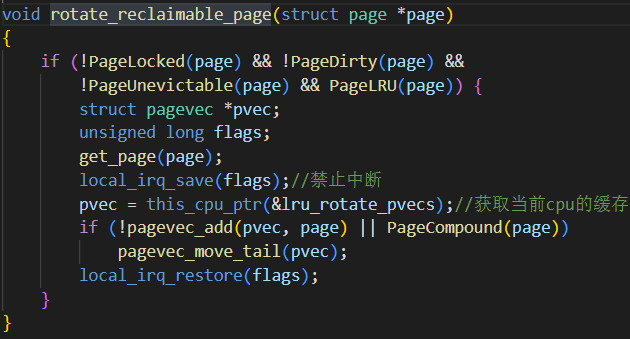
- add\_page\_to\_lru\_list：把page加入到指定lruvec的相应类型的lru链表**头部。**



#### 1.5 rotate\_reclaimable\_page

用于脏页回写。如果在页面回收的时候需要脏页回写，**回写结束后**，如果reclaim被置位，将当前页（在链表中），放到非活跃链表尾。这样当下次淘汰该页面的时候就正常淘汰掉了。

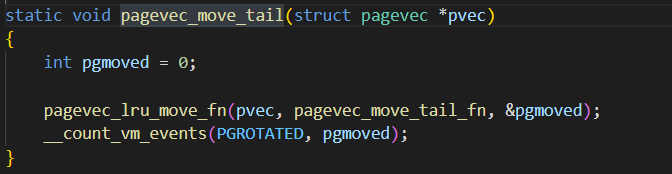
执行过程的逻辑和lru\_cache\_add完全相同。即先把要操作的页加入缓存，如果缓存满了或者是复合页就执行操作。



#### 1.6 pagevec\_move\_tail

操作指示函数。当lru\_rotate\_pvecs缓存满或者遇到复合页，执行，把缓存中的所有页从所在的lru链表头移到链表尾。

和\_\_pagevec\_lru\_add一样，不过这里的pagevec\_lru\_move\_fn的参数改成了pagevec\_move\_tail\_fn。

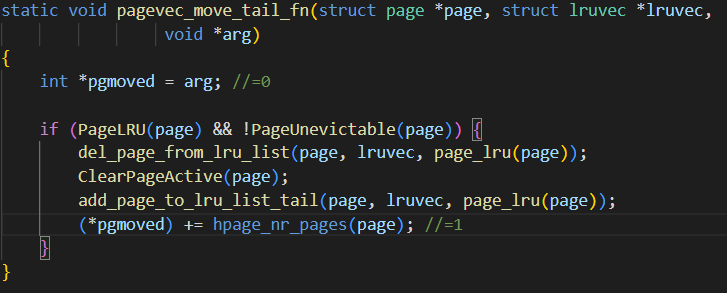


#### 1.7 pagevec\_move\_tail\_fn

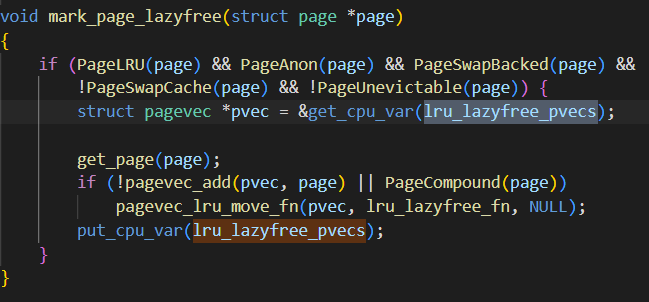
先把page从lru链表中删除

再把page的active位清除（即改为inactive）。这里如果原本page在活跃链表，则也同样加入不活跃链表尾。（因为rotate操作只有回写的时候调用，而回写只有页面需要淘汰的时候发生，而页面淘汰首先淘汰不活跃链表。所以当回写完成的时候将页面加入不活跃链表的尾。）

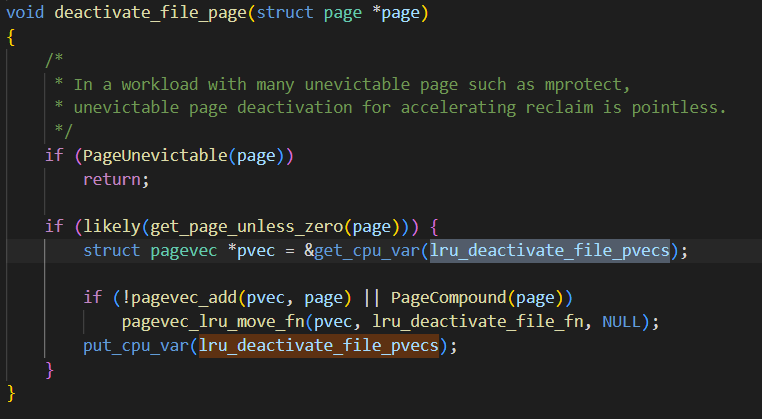
加入相应的不活跃链表尾部。



#### 1.8 mark\_page\_lazyfree



#### 1.9 deactivate\_file\_page



\*1.8 1.9 作用分别是把匿名页或文件页从活跃链表移到不活跃链表，其逻辑和执行函数和之前的两种操作一样。

#### 1.10 mark\_page\_accessed

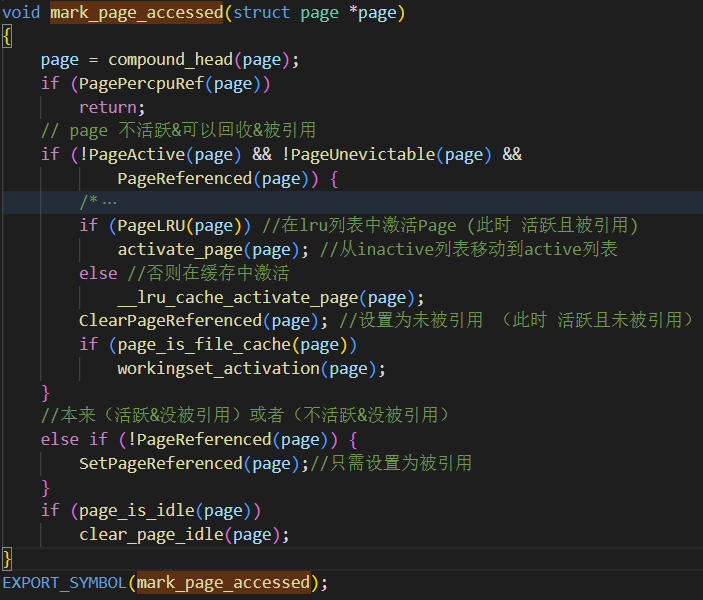
功能：把被page设置为活跃的。

二次机会法。对于page的状态设为三种。并不会直接把不活跃且没被引用的页面标记为活跃，而是先标记为不活跃且被引用，下一次调用该函数再标记为活跃但是会再次标记为不被引用，如果之后又引用了该页面，彻底改为活跃且被引用。

这样的目的是防止对页面的突然引用把一个在inactive lru链表页面直接设置为活跃，然后移动到active的lru链表中；但是后来又不再使用这个页面或者很少访问这个页面，则这个页会很久不被淘汰出去。所以先不直接设置为active而且只设置为referenced作为一个缓冲。

\*\*但是我没找到这个函数在哪被调用...貌似在shrink里没有用到这个函数，而是直接调用了SetPageReferenced





### 内存回收

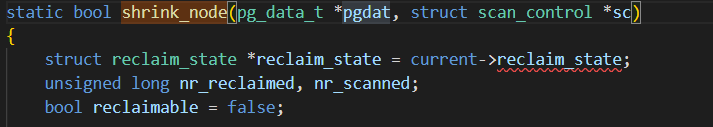
这部分函数并不在swap.c里，主要在vmscan.c里，但是是实际上执行内存回收的函数，只作简要介绍主要功能。

#### shrink\_node

制定某种策略来扫描节点的lruvec中的lru链表，并且确定可以淘汰的页面。

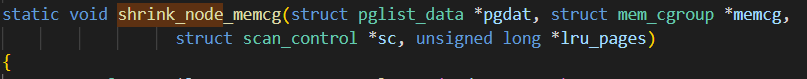
总体的策略是：1优先扫描不活跃的链表，确定可以淘汰的页面；2如果淘汰这部分的页面不够，则转去扫描活跃链表，把其中不太常用的部分放到不活跃链表；3之后再重复第一步，扫描不活跃链表。

除了活跃和不活跃链表之间的切换，还有文件页和匿名页之间的平衡，也就是哪个多扫描的就多。



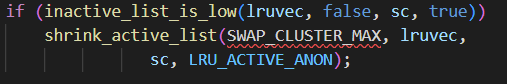
#### shrink\_node\_memcg

shrink\_node调用的最重要的函数，对node中的lruvec进行回收



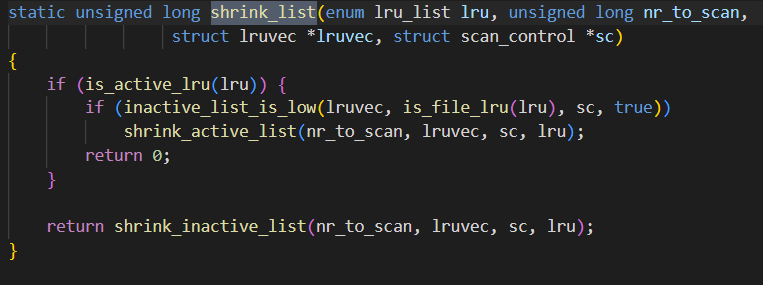


如果不活跃链表比较少，扫描活跃链表



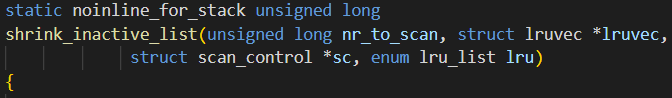
#### shrink\_list

如果活跃链表页面比较少，则扫描活跃链表；否则优先扫描不活跃链表。

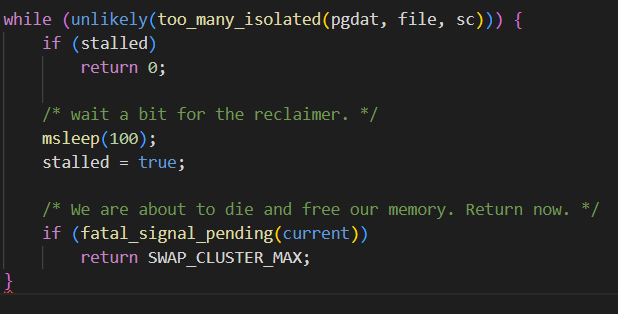


#### shrink\_inactive\_list

进行非活跃链表的回收，并返回回收的页面数。



如果还有别的进程对node进行内存回收，也就是有太多被隔离的页，则会先等待100ms。

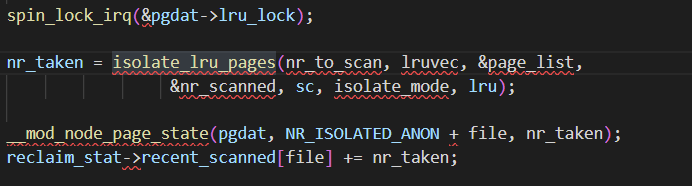


把lru操作缓存中的页面写入相应的lru链表



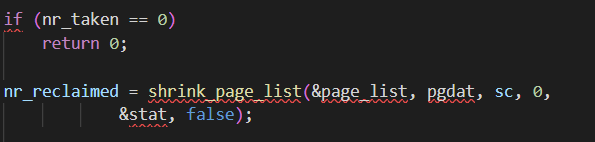
上锁

从非活跃lru链表尾开始隔离出要回收的链表



如果没有隔离出来的页面，说明没有页面能回收，返回0

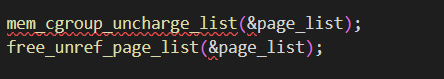
否则进入shrink\_page\_list进行页面回收



page\_list里不能完成回收的页面再写回lruvec对应的lru链表里



最后剩下在page\_list里的页面回收掉

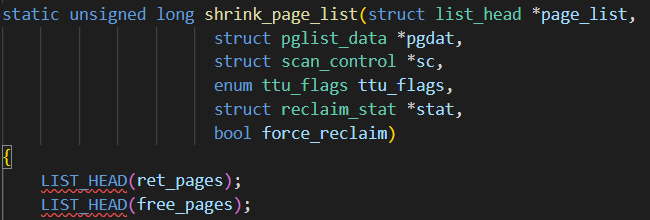


#### shrink\_page\_list

回收page\_list里的页面

ret\_pages临时记录了暂时不能回收的页面，后面进行统一处理

free\_pages记录可以回收的页面



其中对于不同页面的回收不同。

1. 干净页直接unmap解除映射之后，文件页释放huffer之后释放，匿名页可以直接释放。
2. 脏文件页，先unmap解除映射，异步回写并且加入非活跃文件链表头，回写完放入非活跃文件链表尾部，快速回收。
3. 脏匿名页，先加入swapcache，unmap解除映射，异步回写并且加入非活跃匿名链表头，回写完放入非活跃匿名链表尾部，从swapcache中去除，快速回收。

## 调用和依赖关系图

见pdf: **lab2函数调用liuchengt.pdf**

## 实验部分

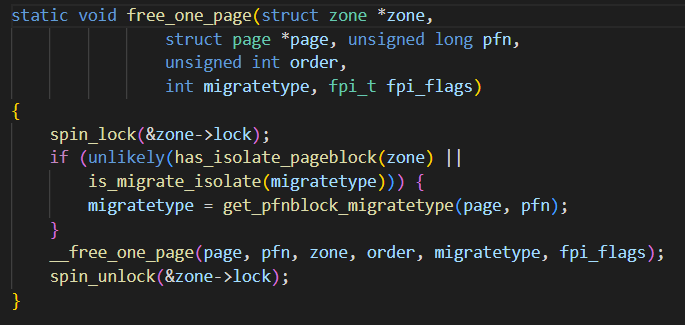
### 实验思路

实验目的：标记换出的页面。

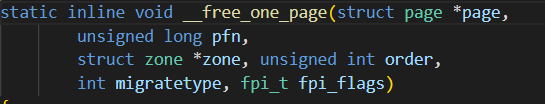
由之前的分析，openeuler页面淘汰分为很多不同的类型，而且在不同触发下函数也不同。想要记录每个淘汰页面的页帧号pfn，需要找到其中的共性或者在每个部分分别进行Log。

通过阅读代码，我发现内核淘汰页面最后会归结为淘汰单个页面的指示函数free\_one\_page和执行函数\_\_free\_one\_page

指示函数：free\_one\_page：



执行函数：\_\_free\_one\_page：

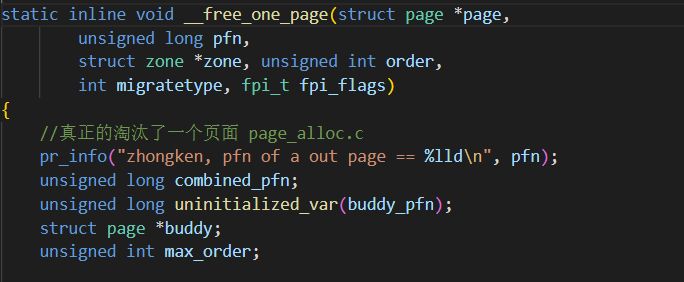


### 代码和截图

#### 代码修改

代码见文件：**page\_alloc.c** 840行左右开始的\_\_free\_one\_page

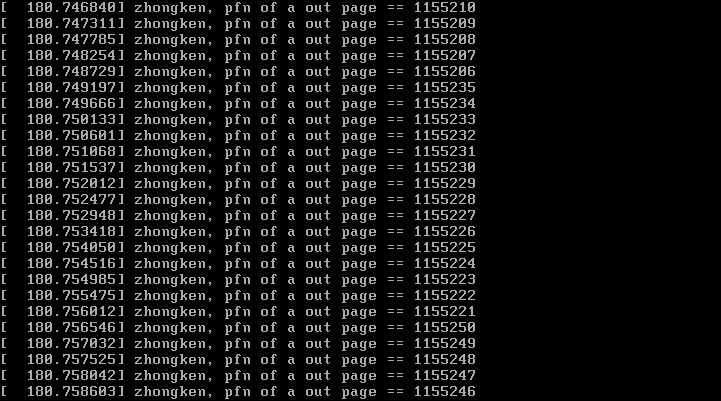
所以根据上面的分析，修改的代码就简单了，我只需要修改最后的执行函数，让其每次在淘汰页面的时候向log文件中输出一下淘汰页面的页帧号就可以了。修改的代码如下：



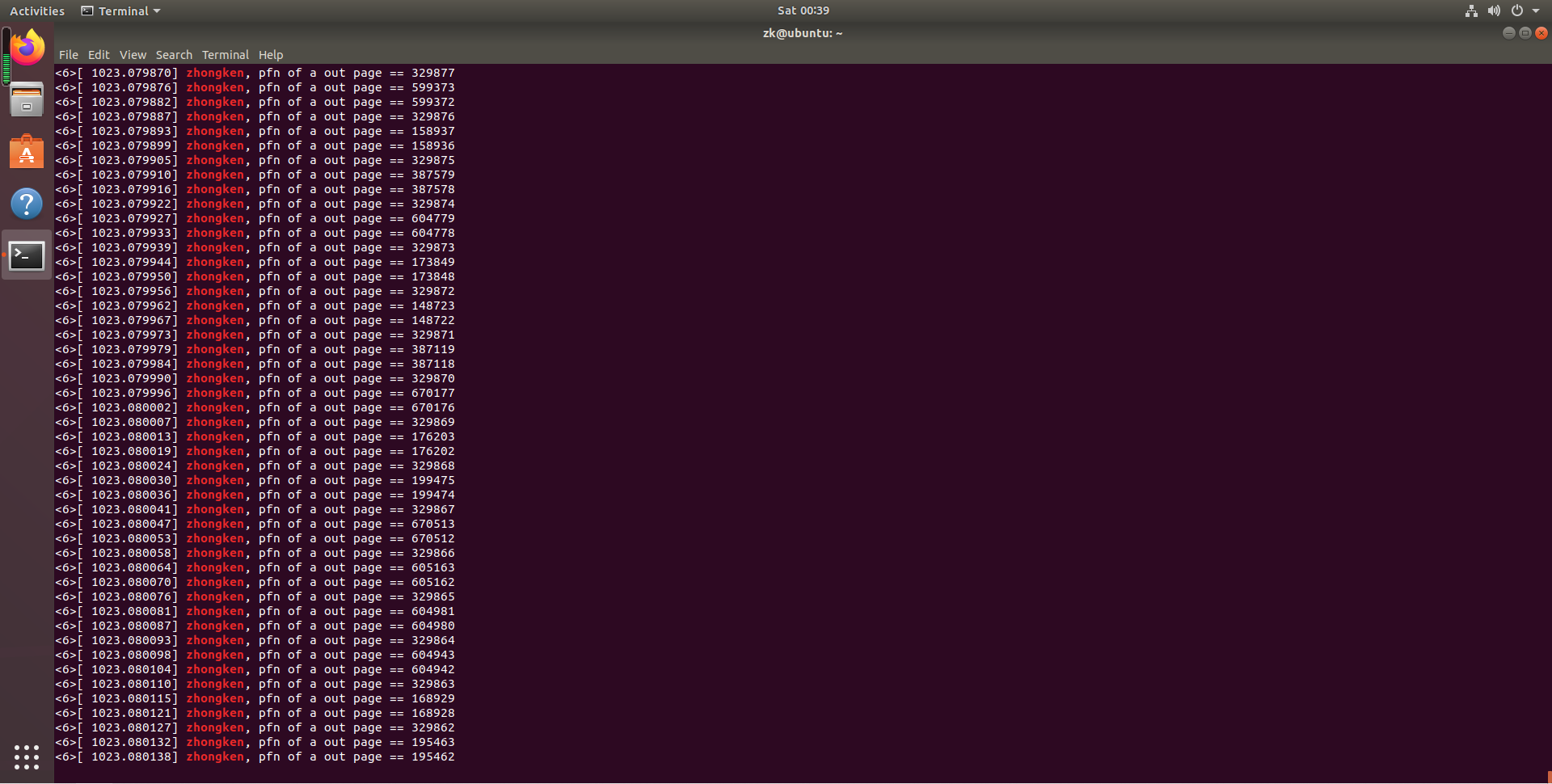
这里我使用了内核封装好的pr\_info来输出log信息，这样在内核启动的时候可以直接看到输出信息，也能在系统运行的时候从/proc/kmsg中找到。为了方便后续查找，这里用我的名字zhongken来标记，后续只需要grep一下就找到了所有的淘汰的页帧号pfn。

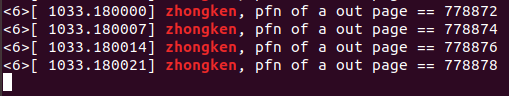
#### 2.2 实验结果截图

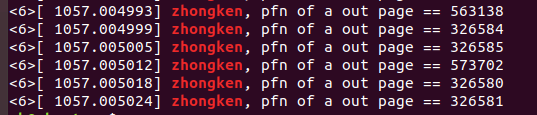
内核启动的时候自然涉及到大量的页面换进换出，下面截图为修改完代码并且编译之后打印的淘汰的页面的pfn。



由于pr\_info 的封装原因，打印的内容会统一放在log文件/proc/kmsg。所以系统运行的时候使用指令 sudo cat /proc/kmsg |grep zhongken动态查看运行过程中的淘汰页面的页帧号。







可以看到随着运行时间而动态淘汰页面并且记录页帧号。实验结束。