

NIS2334 实验报告

实验二必做 + 实验二选做

赶大作业悲伤人士

1 实验二必做

1.1 概述

openEuler 系统采取 LRU 算法进行页面淘汰,其 LRU 算法的实现主要是基于一对双向列表,active list 和 inactive list,其中较活跃页面放在 active list 上,不活跃页面放在 inactive list。在内存严重不足或 kswaped 扫描出操作系统内存低于阈值时,内核会开始页面回收,通过扫描器和一系列检查、准备操作做,从 inactive list 尾部回收页面。LRU 回收方法主要有快速内存回收、直接内存回收和 kswaped 内存回收三种。

1.2 数据结构

1.2.1 扫描器结构体

主要由 scan_control 结构体用于控制和调整页面回收过程的参数和信息,部分数据项如下:

```
// mm\vmscan.c
struct scan_control {
    unsigned long nr_to_reclaim; //控制总回收页框数

    // 控制匿名页lru和文件页lru链表的扫描平衡
    unsigned long anon_cost;
    unsigned long file_cost;

    unsigned int may_writepage:1; // 是否回写
    unsigned int may_unmap:1;
    unsigned int may_swap:1; //是否将匿名页放入swap区

    s8 priority; //每次扫描页框数
    unsigned long nr_scanned; // 已经扫描的页框数
    unsigned long nr_reclaimed; // 已经回收的页框数
};
```

其中三个标识符分别控制:

1. writepage: 记录是否可以回写到磁盘。若不能,则不能处理脏页(数据需要写回同步)或匿名页(没有对应文件数据)
2. unmap: 是否可以解除映射操作。若不能,则只能处理非映射页。
3. swap: 是否可以写入 swap 分区。若不能,则不能处理已使用的匿名页。

1.2.2 LRU 基本内存结构

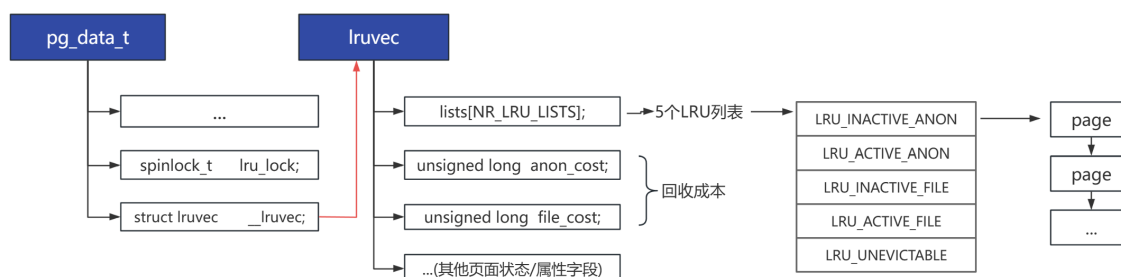


图 1: 部分结构示例图

如图 1 所示，pglist_data 用于存储页面列表信息，lruvec 结构用于存储 lru 列表信息，其中 lists[NR_LRU_LISTS] 为 5 个 lru 列表的头结点，5 个 lru 列表的定义如下。

```
enum lru_list {
    LRU_INACTIVE_ANON = LRU_BASE,
    LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
    LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
    LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
    LRU_UNEVICTABLE,
    NR_LRU_LISTS
};
```

在后续对 page 的操作中，只要获取目标页面的 Active/Inactive 和 File/Anon 值，就可以定位到目标 lru 列表。

每一个 lru 链表头结点都链接了 page 结构链表。page 结构存储单个页面，是页面回收的基本操作单位，主要分为文件页（对应磁盘文件读到 memory 后暂存的页）和匿名页（没有文件背景的内存，存储如 stack，heap 等数据）。page 结构中重要的数据项有：

```
struct page{
    unsigned long flags; //页面状态，用于异步更新
    union{
        struct{
            struct list_head lru; //所在lru列表头
            ...
        };
        ...
    }
}
```

其中与 LRU 算法相关的 flags 状态有：

- PG_active: 页活跃标志
- PG_referenced: 页是否被引用
- PG_lru: 页是否在 lru 链表中

- PG_mlocked: 页是否被内存锁定
- PG_swapbacked: 页是否依赖 swap

1.2.3 LRU 操作结构

为了更细粒度地控制自旋锁范围、更高效地进行页面操作,openEuler 在”include/linux/pagevec.h”中定义了 pagevec 结构体来缓存一定数量的 page, 并进一步定义了 lru_rotate 和 lru_pvecs 作为 lru 链表操作对应的页面缓存集。其主要定义如下。

```
#define PAGEVEC_SIZE 15
struct pagevec {
    unsigned char nr; // 当前page数量
    struct page *pages[PAGEVEC_SIZE];
};

struct lru_rotate {
    local_lock_t lock;
    struct pagevec pvec;
};

struct lru_pvecs {
    local_lock_t lock;
    struct pagevec lru_add; // 添加到lru列表
    struct pagevec lru_deactivate_file; // 文件页：从活动列表移动到非活动列表
    struct pagevec lru_deactivate; // 从活动列表移动到非活动列表
    struct pagevec lru_lazyfree; // 匿名页：从活动列表移动到非活动列表
#ifdef CONFIG_SMP // SMP机器操作
    // 将非活动页移动到活动页的缓存
    struct pagevec activate_page;
#endif
};
```

当需要对 lru 链表进行操作时, page 将被加到对于操作的缓存集中, 在缓存集满或者其他适当的时候将缓存集中的页面进行一次性处理操作。

1.3 算法流程与函数

整个算法流程可以分为局部的 LRU 操作实现、整体内存回收实现和不同内存回收方法的实现三部分。

1.3.1 LRU 链表操作缓存函数

LRU 主要的函数操作在表 1 中列出。

其中 deactivate_file_page 和 deactivate_page 处理没有被进程映射的页面, 而 ake_page_lazyfree 函数处理正在回写到 swap 分区的匿名页, rotate_reclaimable_page 将写回的页面加入列表尾部。以

上四个函数都针对页面属性进行了回收优化，通过控制写回 inactive 页面的头部/尾部来延迟/加快不同属性页面的回收。

lru 操作缓存列表	主要函数	作用
lru_add	lru_cache_add	将 cpu 中的页面缓存添加至 lru 列表
lru_deactivate_file	deactivate_file_page	将不经常访问文件页从 active 列表添加到 inactive 列表
lru_deactivate	deactivate_page	将不经常访问页面从 active 列表添加到 inactive 列表
lru_lazyfree	make_page_lazyfree	将不经常访问匿名页从 active 列表添加到 inactive 列表
lru_rotate	rotate_reclaimable_page	将回写完的页面加入 inactive 列表尾部

表 1: lru 列表操作主要函数

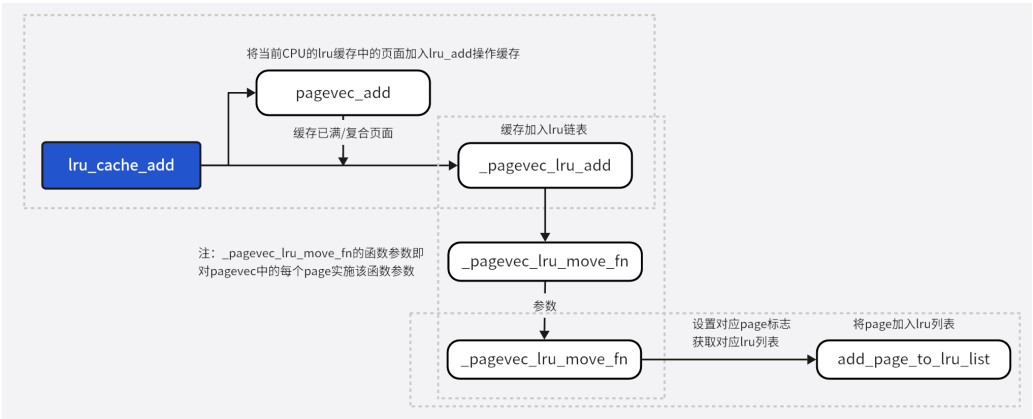


图 2: lru_cache_add callmap

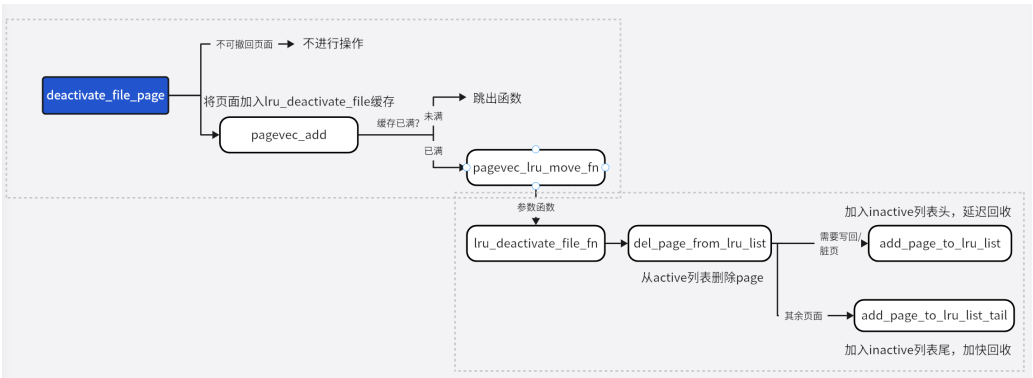


图 3: deactivate_file_page callmap

图 2-3 为表中函数部分函数¹的 callmap 与注解。观察函数调用过程，可以发现其基本过程可以分为两种：

1. 获取 page-> 加入对应 pagevec -> 若满，则直接调用 add/move 函数
2. 获取 page-> 加入对应 pagevec -> 若满，则调用 move_fn，控制不同函数参数，对 pagevec 上的页面做对应操作（在此处为从原列表上删除，加入新列表）

除此以外，与 LRU 算法相关的重要函数还有 `mark_page_accessed`。为了更精细地控制 active 列表和 inactive 列表上页面的转移，该系统设置了四种状态：

- inactive,unreferenced
- inactive,referenced
- activate,unreferenced
- activate,referenced

`mark_page_accessed` 就是用于控制页面状态转换的函数。

1.3.2 页面回收函数

主要定义在 `vmscan.c` 中，整体函数依赖关系如图 4 所示。

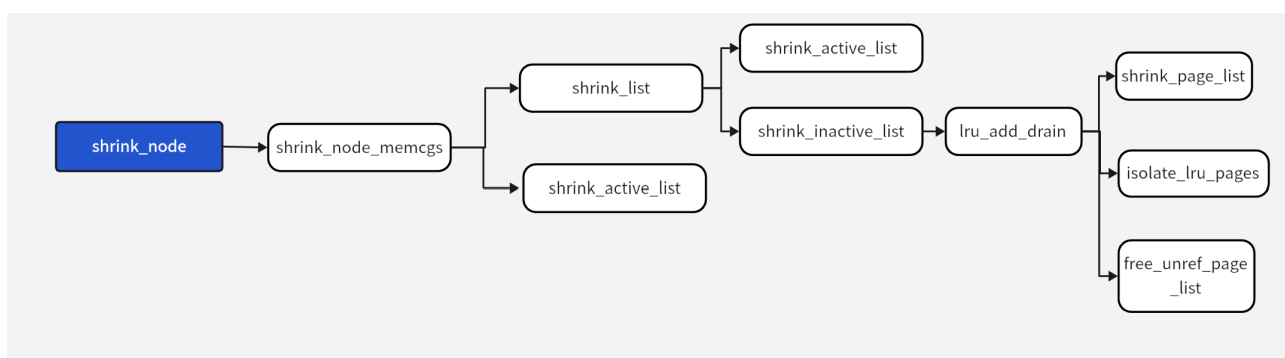


图 4: `lru_cache_add` callmap

核心函数为 `shrink_node`。该函数基本原理如图 5 所示。

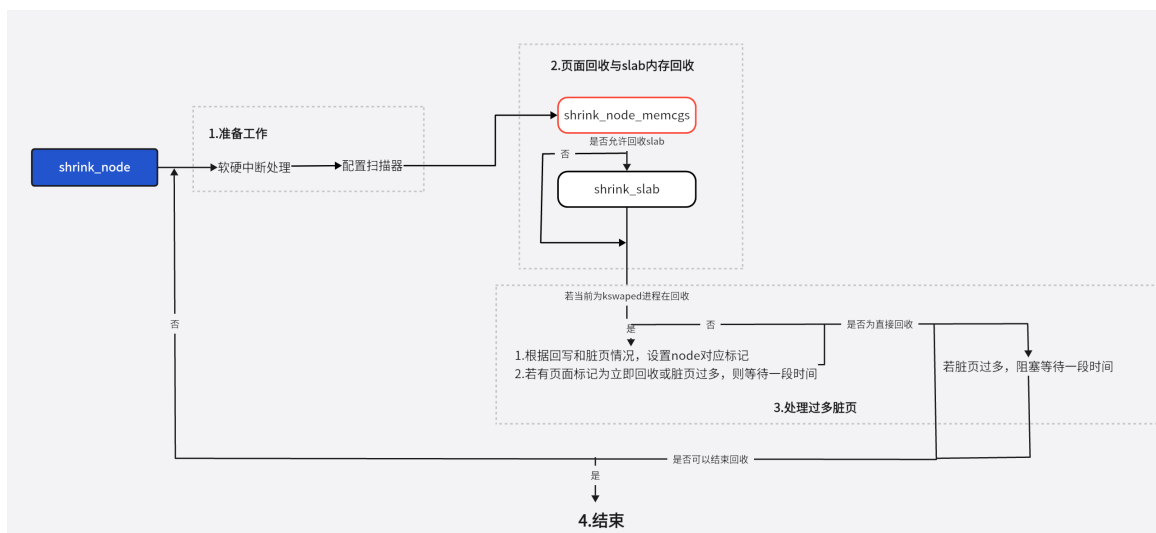


图 5: `lru_cache_add` callmap

¹由于 `deactivate_page`、`mark_page_lazyfree` 函数和 `deactivate_file_page` 函数基本调用流程和操作相似，`rotate_reclaimable_page` 函数和 `pagevec_add` 函数基本调用流程和操作相似，将不绘制 `deactivate_page` 函数、`mark_page_lazyfree` 函数和 `pagevec_add` 函数的 callMap。

可以看出 shrink_node 函数会在运行过程中进行 node_memcgs 和 slab 内存的循环回收，每一次回收前后都会进行准备和优化过程，完成后会判断回收是否结束。

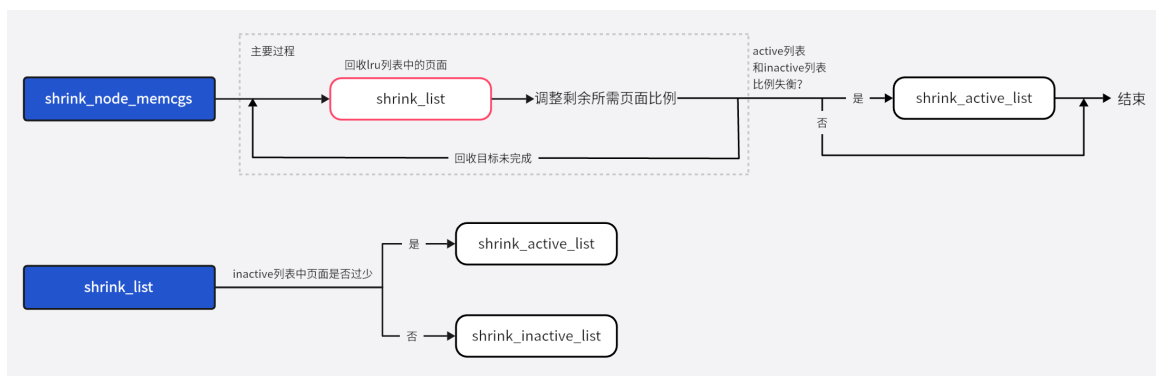


图 6: shrink_node_memcgs & shrink_list callmap

shrink_node_memcgs 函数用于回收 node 中的 lru。其主要代码段为：

```
static void shrink_node_memcg(struct pglist_data *pgdat, struct mem_cgroup *memcg, struct
    scan_control *sc, unsigned long *lru_pages)
{
    ...
    while (nr[LRU_INACTIVE_ANON] || nr[LRU_ACTIVE_FILE] || nr[LRU_INACTIVE_FILE]){
        for_each_evictable_lru(lru){
            \dots
            nr_reclaimed += shrink_list(lru, nr_to_scan, lruvec, sc);
        }
        \dots/* 调整文件页和匿名页比例的代码 */
    }
    \dots
    if (inactive_list_is_low(lruvec, false, sc, true)){
        shrink_active_list(SWAP_CLUSTER_MAX, lruvec, sc, LRU_ACTIVE_ANON);
    }
}
```

可以看出该函数会先不断循环，执行 shrink_list 函数来回收页面并调整文件页和匿名页比例，直至回收目标完成或目标无法达到，退出循环后，若 inactive 列表页面过少²，则会执行 shrink_active_list 进行平衡。

shrink_list 函数用于回收或调整 lru 列表中的页面。

若 lru 参数为 active_list，则执行 shrink_active_list 调整列表平衡；若 lru 参数为 inactive_list，则执行 shrink_inactive_list 进行非活跃页面的回收。

shrink_active_list 函数用于缩减 active 列表中的页面数量，从图 7 可以看出，该函数会从 active 列表尾开始筛选一定数量当前回收区中、可正常获取的页面，放入 l_hold 列表作为候选，随后会选择匿名页或非缓存执行代码的文件页，将其从 active 列表放入 inactive 列表。

shrink_inactive_list 函数用于回收 inactive 列表中的页面，主要调用 shrink_page_list 和

²在页面回收过程中，active 列表与 inactive 列表的比例需要尽可能接近 3:1（由函数 inactive_list_is_low 判定）

free_undef_page_list 函数实现。

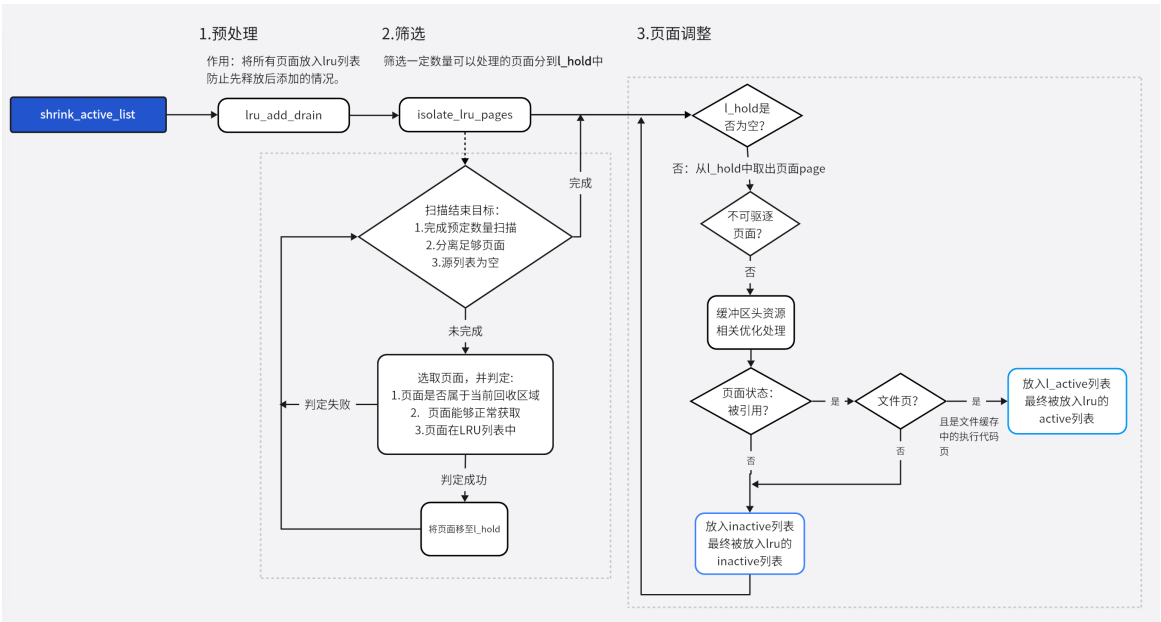


图 7: shrink_active_list 流程图

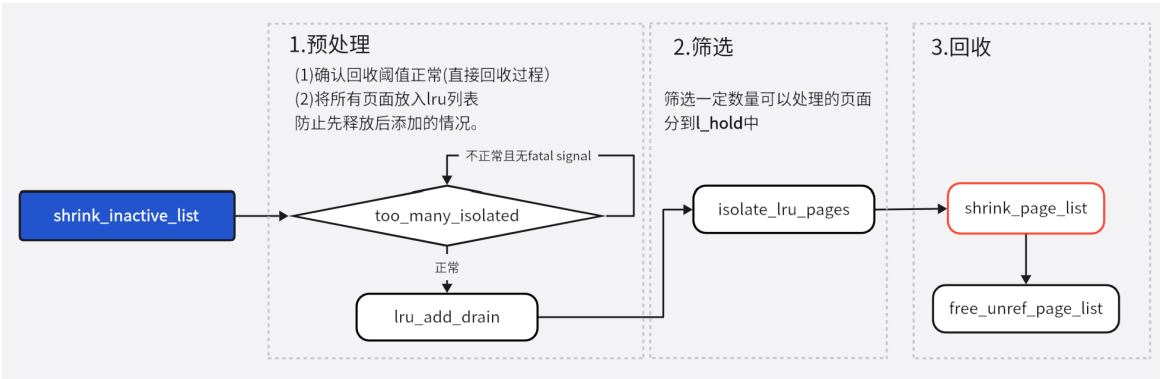


图 8: shrink_inactive_list 流程图

shrink_page_list 函数用于遍历页面列表，尝试释放页面进行回收。由图 9 可得，该函数会检查：

- 页是否可回收
- 页的状态，包括是否脏、是否在写回等
- 页的引用情况
- 页的类型

根据检查结果决定是否激活页、保留页、回收页或执行其他操作，如取消映射、释放缓冲区等。

页面回收的主要入口点有：do_try_to_free_pages 和 do_swapcache_reclaim。

前一个为直接回收页面的主要函数，会调用 shrink_zones（其中会进一步调用 shrink_node）来进行页面回收。后一个为回收交换缓存页面的主要入口点，会调用 reclaim_swapcache_pages_from_list

(其中会进一步调用 `shrink_page_list`) 实现页面回收。

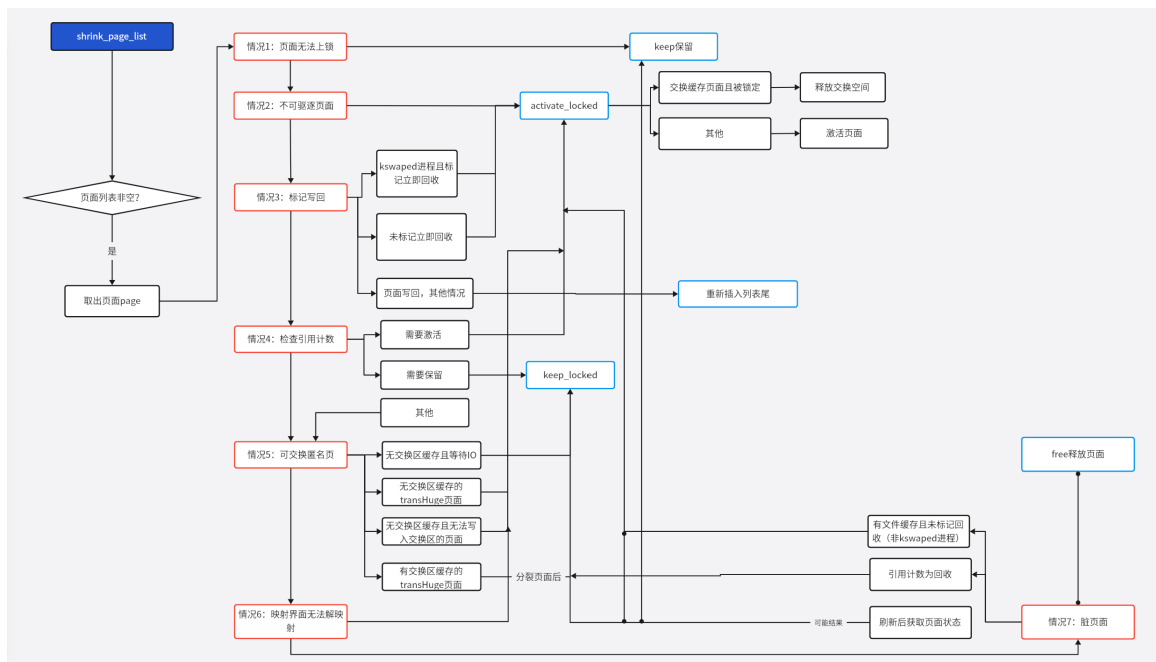


图 9: shrink_page_list 流程图

1.3.3 三种回收方式

在该操作系统中，一共有三种页面回收方法：快速内存回收、直接内存回收和 kswaped 线程回收。后两种方式由操作系统的内存状态（内存使用率）控制。

在操作系统中定义了三个水位线：pages_high, pages_low 和 pages_min。pages_high 表示当前操作系统内存状态是良好的，此时 kswaped 线程处于 sleep 状态。当内存使用率触及 pages_low，就意味着内存不足，操作系统会开始内存回收，由进程 kswaped 周期性检查并回收内存。当内存使用率触及 pages_min，即内存严重不足，操作系统会开始直接内存回收。

1.4 实验实现

实验目的：记录每次发生页面淘汰时淘汰的页面


```
mm > C page_alloc.c >  __free_one_page(page *, unsigned long, zone *, unsigned int, int, fpi_t)
842 static inline void __free_one_page(struct page *page,
940 |         }
941 |     }
942 |
943 |     list_add(&page->lru, &zone->free_area[order].free_list[migratetype]);
944 out:
945 |     pr_info("Success to free page,whose pfn is:%lld\n",pfn); //print pfn of freed page to log
946 |     zone->free_area[order].nr_free++;
947 }
```

图 10: 代码修改: 添加页面淘汰信息

1.4.1 解决思路

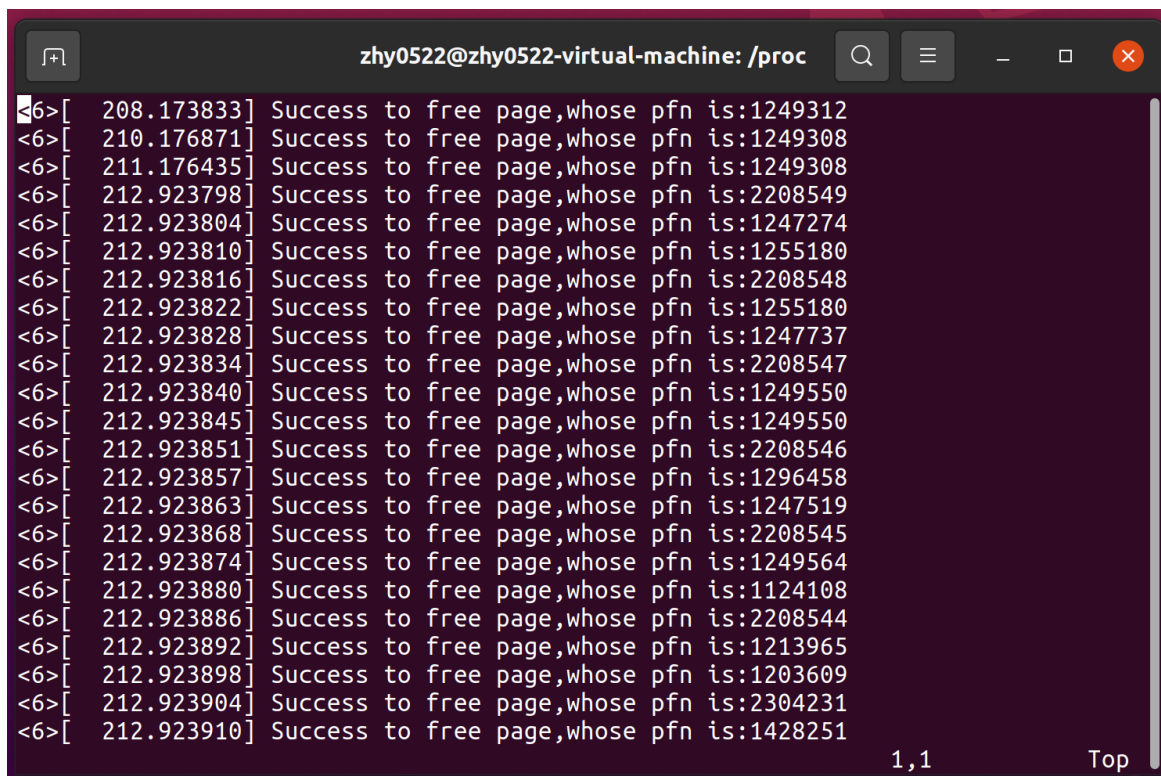
操作系统中的页面有唯一标志符 pfn，为页面在 mem_map 结构中的下标。因此，只需找到页面淘汰的底层释放函数，添加淘汰页面的信息至系统日志。

页面淘汰底层函数为 mm/page_alloc.c/___free_one_page。该函数在成功释放页面之后增加空闲页面的计数，因此修改代码，将释放信息函数写于更新计数代码行前，如图 10 所示。

其中 print_info 函数用于将信息输出到 log 中，输出文件为 /proc/kmsg。

1.4.2 实验结果

打开 /proc/kmsg 文件，看到文件中成功输出了回收页面的 pfn 信息，则实验成功。



```
zhy0522@zhy0522-virtual-machine: /proc
[6>[ 208.173833] Success to free page,whose pfn is:1249312
<6>[ 210.176871] Success to free page,whose pfn is:1249308
<6>[ 211.176435] Success to free page,whose pfn is:1249308
<6>[ 212.923798] Success to free page,whose pfn is:2208549
<6>[ 212.923804] Success to free page,whose pfn is:1247274
<6>[ 212.923810] Success to free page,whose pfn is:1255180
<6>[ 212.923816] Success to free page,whose pfn is:2208548
<6>[ 212.923822] Success to free page,whose pfn is:1255180
<6>[ 212.923828] Success to free page,whose pfn is:1247737
<6>[ 212.923834] Success to free page,whose pfn is:2208547
<6>[ 212.923840] Success to free page,whose pfn is:1249550
<6>[ 212.923845] Success to free page,whose pfn is:1249550
<6>[ 212.923851] Success to free page,whose pfn is:2208546
<6>[ 212.923857] Success to free page,whose pfn is:1296458
<6>[ 212.923863] Success to free page,whose pfn is:1247519
<6>[ 212.923868] Success to free page,whose pfn is:2208545
<6>[ 212.923874] Success to free page,whose pfn is:1249564
<6>[ 212.923880] Success to free page,whose pfn is:1124108
<6>[ 212.923886] Success to free page,whose pfn is:2208544
<6>[ 212.923892] Success to free page,whose pfn is:1213965
<6>[ 212.923898] Success to free page,whose pfn is:1203609
<6>[ 212.923904] Success to free page,whose pfn is:2304231
<6>[ 212.923910] Success to free page,whose pfn is:1428251
1,1 Top
```

图 11: 必做实验结果图

2 实验二选做

2.1 基本目标与思路

实验目标为设计数据结构，实现 FIFO 算法。设最大缓存页面数为 10，并采用数组记录页面，在每一次分配新页面的时候检查数组，若数组已满，则输出最先分配页面的 pfn 值，并将当前新页面入队。

2.2 具体实现

首先，在 `page_alloc.c` 中定义存储 FIFO 的数组结构。`count` 为数组中元素个数，`front` 和 `rear` 为头尾元素索引值，`lock` 为自旋锁，使得数组内容能在临界区内被修改。

```
//define the struct of FIFO
#define FIFO_size 10
typedef struct {
    unsigned long array[FIFO_size];
    int front;
    int rear;
    int count;
    spinlock_t lock;
} FIFO;

static FIFO fifo; //Define global array variables
```

其次，实现该数组的 `init` 函数，并且在 `page_alloc` 初始化时初始化数组，其中 `page_alloc_init` 为 `cpu` 加载时 `page_alloc` 的初始化函数，因此对其作一行修改。

```
static void initFIFO(FIFO *fifo) {
    fifo->front = 0;
    fifo->rear = -1;
    fifo->count = 0;
    spin_lock_init(&fifo->lock); //built-in init function in Linux
}

void __init page_alloc_init(void)
{
    int ret;
    initFIFO(&fifo); //Modified here.Init FIFO array
    ret = cpuhp_setup_state_nocalls(CPUHP_PAGE_ALLOC_DEAD,
        "mm/page_alloc:dead", NULL,
        page_alloc_cpu_dead);
    WARN_ON(ret < 0);
}
```

考虑到实验目标，则只需实现队列的入队，并且在队列满时出队即可，则定义 `enqueue` 函数如下。在页面入队/出队时，会调用 `printk` 函数，将相关内核信息输出至系统日志中。为调试方便，此处将消息等级处理为 `KERN_DEBUG`。

```
static bool isFull(FIFO *fifo) {
    return fifo->count == FIFO_size;
}

//add pfn of the new_page to the tail of the queue
static void enqueue(FIFO *fifo, unsigned long pfn) {
    unsigned long flags;
    //enter critical
    spin_lock_irqsave(&fifo->lock, flags);
```

```

//check if the queue is full
if (isFull(fifo)) {
    unsigned long removed_value = fifo->array[fifo->front];
    //print dequeue information
    printk(KERN_DEBUG "Dequeued FIFO_page,whose pfn is %lu\n", removed_value);
    fifo->front = (fifo->front + 1) % FIFO_size;
    fifo->count--;
}
//new page enqueue
fifo->rear = (fifo->rear + 1) % FIFO_size;
fifo->array[fifo->rear] = pfn;
fifo->count++;
//exit critical
spin_unlock_irqrestore(&fifo->lock, flags);
//print enqueue information
printk(KERN_DEBUG "add FIFO_page ,whose pfn is %lu\n",pfn);
}

```

最后，由于分配新页面时调用底层函数为 `get_page_from_freelist`，则只需在该函数成功获取页面并准备返回时调用 `enqueue` 函数即可，具体修改如下：

```

static struct page *
get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags,
                      const struct alloc_context *ac)
{
    \dots
    try_this_zone://label line of getting free page
        page = rmqueue(ac->preferred_zoneref->zone, zone, order,
                      gfp_mask, alloc_flags, ac->migratetype);
        if (page) {
            prep_new_page(page, order, gfp_mask, alloc_flags);

            /*
             * If this is a high-order atomic allocation then check
             * if the pageblock should be reserved for the future
             */
            if (unlikely(order && (alloc_flags & ALLOC_HARDER)))
                reserve_highatomic_pageblock(page, zone, order);
            //Modified here:get pfn of the page and add pfn to array if getting page correctly
            if(page) enqueue(&fifo,page_to_pfn(page));
            return page;
        }
    \dots
}

```

2.3 实验结果

成功编译并替换内核后, 切换到 4.19.0 内核, 并且在终端中输入 `dmesg | grep "Dequeued" | head -n 30` 获取最开始的 30 条输出信息 (过滤入队消息)。

```
zhy0522@zhy0522-virtual-machine:~/Desktop$ dmesg | grep "Dequeued" | head -n 30
[ 89.199026] Dequeued FIFO_page,whose pfn is 1275211
[ 89.199207] Dequeued FIFO_page,whose pfn is 1275330
[ 89.199241] Dequeued FIFO_page,whose pfn is 1971100
[ 89.199265] Dequeued FIFO_page,whose pfn is 1275331
[ 89.199299] Dequeued FIFO_page,whose pfn is 1261874
[ 89.199324] Dequeued FIFO_page,whose pfn is 1970525
[ 89.199431] Dequeued FIFO_page,whose pfn is 1269517
[ 89.199938] Dequeued FIFO_page,whose pfn is 1261875
[ 89.199979] Dequeued FIFO_page,whose pfn is 1269386
[ 89.199986] Dequeued FIFO_page,whose pfn is 1269518
[ 89.200007] Dequeued FIFO_page,whose pfn is 1269387
[ 89.200012] Dequeued FIFO_page,whose pfn is 1269368
[ 89.200063] Dequeued FIFO_page,whose pfn is 1269369
[ 89.200129] Dequeued FIFO_page,whose pfn is 1269362
[ 89.200152] Dequeued FIFO_page,whose pfn is 1269363
[ 89.200157] Dequeued FIFO_page,whose pfn is 1091008
[ 89.200163] Dequeued FIFO_page,whose pfn is 1154874
[ 89.200170] Dequeued FIFO_page,whose pfn is 1269346
[ 89.200184] Dequeued FIFO_page,whose pfn is 1269347
[ 89.200270] Dequeued FIFO_page,whose pfn is 1269316
[ 89.200462] Dequeued FIFO_page,whose pfn is 1269317
[ 89.200550] Dequeued FIFO_page,whose pfn is 1275512
[ 89.200651] Dequeued FIFO_page,whose pfn is 1275513
[ 89.200822] Dequeued FIFO_page,whose pfn is 1268452
[ 89.201365] Dequeued FIFO_page,whose pfn is 1268382
[ 89.201749] Dequeued FIFO_page,whose pfn is 1266331
[ 89.202320] Dequeued FIFO_page,whose pfn is 2299652
```

图 12: 选做实验结果图