

Linux内核中跟踪文件PageCache预读

原创 知书码迹 知书码迹 2021-11-05 11:09

收录于话题

#linux内核 2 #页缓存 1 #内存 2



[点击蓝字](#) / [阅读美文](#)

概述

预读算法预测即将访问的页面，并提前将页面批量读入内存缓存。预读算法可能面临多样化的文件访问模式，如非对齐读，即顺序读取文件但是每次读取的页面偏移量不同、不同进程交织读取、混合读取。本文选择顺序且4K对齐的读取模式，借助GDB跟踪文件缓存在内核中的执行路径，并对该读取模式下的预读算法进行分析。

本文字数4.5K，阅读时长≈4分钟

场景描述

通过用户态调用read系统调用，对文件进行每次4K大小的循环顺序读取，进而通过GDB在内核中观察函数的调用栈以及相关变量的取值变化。编写用户态C程序如下：

```
//read.c
int main()
{
    char c[4096];
    int in = -1;
    in = open("news.txt", O_RDONLY);
    int index=0;
    while (read(in, &c, 4096) == 4096)
    {
        printf("index: %d,len: %ld.\n",index,strlen(c));
        memset(c, 0, sizeof(c));
        index++;
    }
    close(in);
}
```

```
    return 0;
}
```

其中，目标读取文件news.txt的大小为128K，预计全部读完需要循环读取32次。

实验环境

内核版本： linux-4.14.191

在需要debug的内核关键函数前添加编译属性如下，降低代码块的编译等级，防止变量被优化掉。

```
__attribute__((optimize("O0")))
```

QEMU启动命令：

```
qemu-system-x86_64 -kernel ~/linux-4.14.191/arch/x86_64/boot/bzImage -hda ~/busybox-1.32.
```

在启动虚拟机的时候，增加了一个共享磁盘hdb，其中保存了在宿主机中编译好的用户态程序read.out和待读取文件news.txt。

内核启动以后，挂载sdb设备，使用echo命令清空系统缓存。

```
/ # mount /dev/sdb /mnt/
[ 353.150473] random: crng init done
[ 353.239270] EXT4-fs (sdb): recovery complete
[ 353.244787] EXT4-fs (sdb): mounted filesystem with ordered data mode. Opts: (null)
/ # ls mnt/
file.txt    lost+found  news.txt   read       read.c     read.out
/ #
```

```
echo 3 > /proc/sys/vm/drop_caches
```

启动GDB:

```
gdb -ex 'target remote localhost:1234' -ex c ./linux-4.14.191/vmlinux
```

添加断点：

```
(gdb) source gdb.cfg
Breakpoint 2 at 0xffffffff8112fa40: file mm/filemap.c, line 1963.
Breakpoint 3 at 0xffffffff811918c0: file fs/read_write.c, line 391.
Breakpoint 4 at 0xffffffff81191964: file fs/read_write.c, line 403.
Breakpoint 5 at 0xffffffff8112fd8b: file mm/filemap.c, line 2058.
Breakpoint 6 at 0xffffffff8112ff00: file mm/filemap.c, line 2114.
Breakpoint 7 at 0xffffffff8112ff28: file mm/filemap.c, line 2120.
Breakpoint 8 at 0xffffffff8112ff77: file mm/filemap.c, line 2138.
Breakpoint 9 at 0xffffffff811300aa: file mm/filemap.c, line 2184.
Breakpoint 10 at 0xffffffff8112fbae: file mm/filemap.c, line 2001.
Breakpoint 11 at 0xffffffff8112fc39: file mm/filemap.c, line 2011.
Breakpoint 12 at 0xffffffff8112fc8d: file mm/filemap.c, line 2017.
Breakpoint 13 at 0xffffffff81130185: file mm/filemap.c, line 2210.
Breakpoint 14 at 0xffffffff8114023c: file mm/readahead.c, line 171.
```

断点

使用如下GDB命令保存现有断点到文件gdb.cfg：

```
save breakpoints gdb.cfg
```

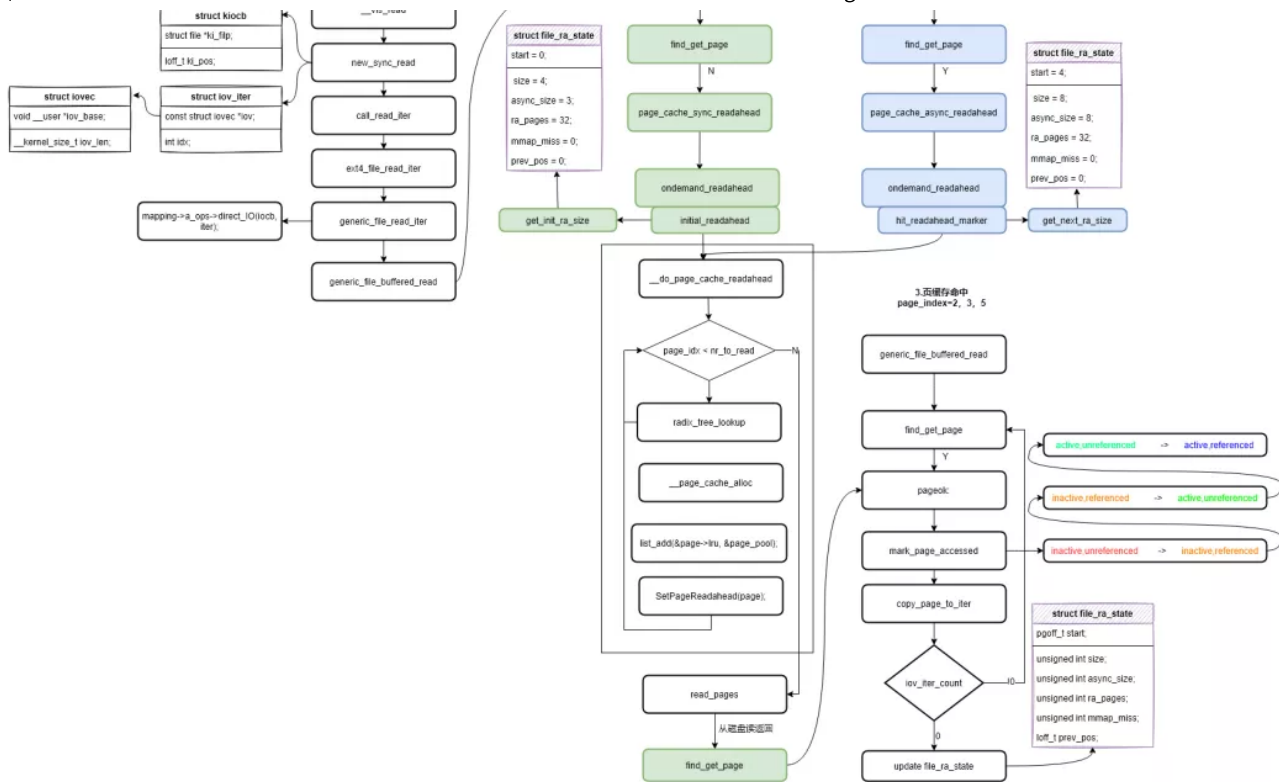
使用如下GDB命令加载以往保存的断点：

```
source gdb.cfg
```

预读函数栈分析

用户态程序执行read系统调用后进入到内核虚拟文件系统层vfs_read函数，如下流程图开始的函数。然后逐层调用，new_sync_read函数中使用struct kiocb结构体包装了struct file结构体，并对当前读取文件的状态进行管理。new_sync_read函数创建struct iov_iter进行内核态-用户态之间数据的拷贝以及记录本次读取长度（len）。而后进入generic_file_read_iter函数进行了direct_IO的判断，即不通过页缓存读取文件数据。如果使用页缓存(PageCache)读取数据就进入了预读的主要处理函数generic_file_buffered_read。





针对上文设计的读取模式，对generic_file_buffered_read函数中的三种执行情况进行分析，整体流程如上图所示，三种执行情况分别如下。

- 首次首部同步预读（第一次读取文件数据）
- 后续异步预读（后续读取，并且命中了预读标识PG_readahead）
- 后续缓存命中读取（缓存命中并且没有进行预读）

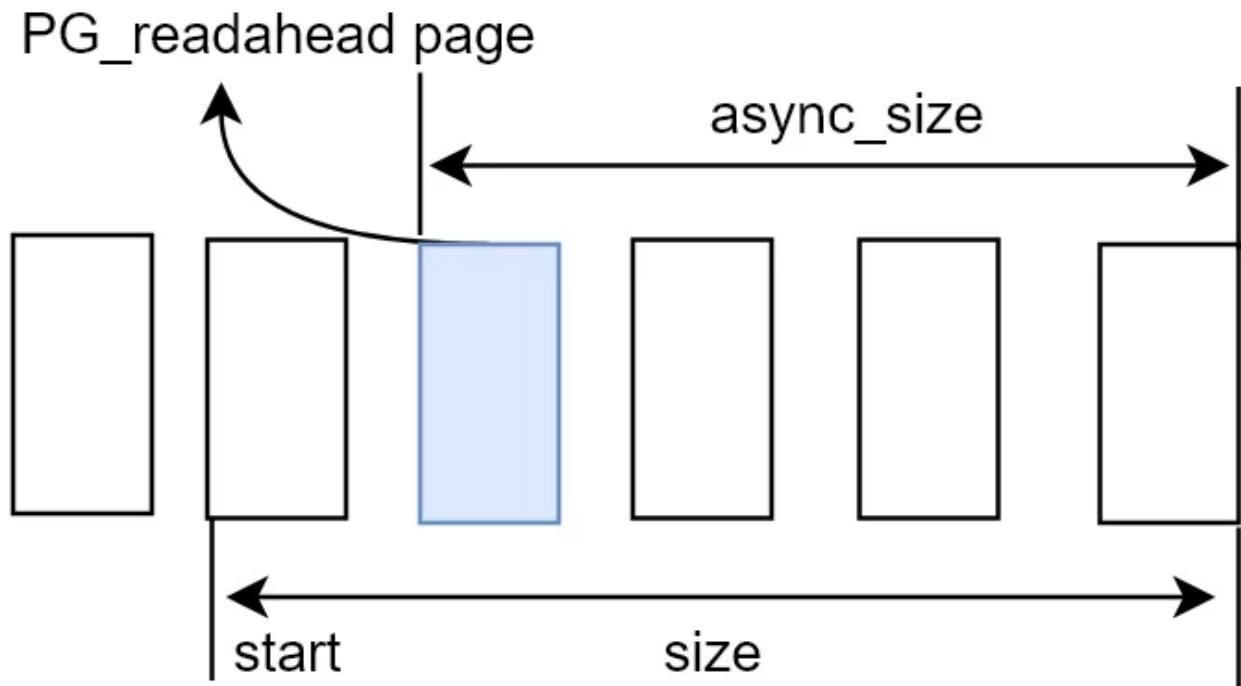
预读窗口

在分析函数执行流之前，首先要介绍一下预读用到的一个核心数据结构预读窗口struct file_ra_state，内核通过该窗口在当前文件读取流中不断后移，实现文件页（page）的预读。

```
struct file_ra_state {
    pgoff_t start; /* where readahead started 当前窗口的第一个page索引，初始0*/
    unsigned int size; /* # of readahead pages 当前窗口的页面数量，值为-1表示预读临时关闭，0表示当
    unsigned int async_size; /* do asynchronous readahead when there are only # of pages a
    unsigned int ra_pages; /* Maximum readahead window 预读窗口最大页面数量。0表示预读暂时关闭，
    unsigned int mmap_miss; /* Cache miss stat for mmap accesses mmap预读命中率。初始0*/
    loff_t prev_pos; /* Cache last read() position Cache中最近一次读位置。初始-1*/
}
```

};

该数据结构中几个的成员的关系如下：



预读窗口

页索引（page_index）为（size - async_size）的页被标示为PG_readahead，表示用户态程序读到该页时需要进行下一次预读，因此async_size的大小决定了当前窗口进行下一次预读并后移的时机。

首次首部同步预读

QEMU中执行read.out程序，程序在new_sync_read函数中的断点停下，打印当前读取的文件名称和读取的长度，确定本次是在读取news.txt文件数据，如下图所示，读取的文件名称(d_iname)为“news.txt”，长度（len）为4096。

```
Breakpoint 3, new_sync_read (filp=0xffff88800679ea00, buf=0x7ffe6d52e150 "", len=4096, ppos=0xfffffc900000dbf10) at fs/read_write.c:391
(gdb) p *filp.f_path.dentry->d_iname@10
$13 = "news.txt\000"
(gdb) p len
$14 = 4096
```

待读取文件名称

继续执行程序到预读主要处理函数generic_file_buffered_read。该函数首先会获取当前读取文件的struct file，以及本次读取数据在文件内的偏移量loff_t *ppos。从struct file中获取初始预读窗口，初始值只有ra_pages=32，表示窗口最大为32个page，prev_pos=-1表示文件还没有读取过，如下图GDB打印结果所示。

```
struct file_ra_state *ra = &filp->f_ra;
```

```

    (ab) p *ra
    = {
        start = 0,
        size = 0,
        async_size = 0,
        ra_pages = 32,
        mmap_miss = 0,
        prev_pos = -1
    }

```

预读窗口

接下来generic_file_buffered_read函数计算文件内相关页索引，以及偏移量，用于计算读取的次数，读取的位置进行读取状态、方式的判断。页索引（index）初始值为0，表示文件中的第一页数据：

```

//页索引
index = *ppos >> PAGE_SHIFT;
//上次页索引
prev_index = ra->prev_pos >> PAGE_SHIFT;

//上次页内偏移量
prev_offset = ra->prev_pos & (PAGE_SIZE-1);

//结束页下标，例如pos读到第1页的数据，则last_index=2
last_index = (*ppos + iter->count + PAGE_SIZE-1) >> PAGE_SHIFT;
//PAGE_MASK是12个0，本次页内偏移量
offset = *ppos & ~PAGE_MASK;

```

接下来generic_file_buffered_read函数调用find_get_page，该函数会从该文件地址空间（address_space）中尝试读取页索引（index）对应的页（page）。因为是首次读取，该文件的数据页并不会在PageCache中找到，因此会执行同步预读函数page_cache_sync_readahead。该函数主要进行两次判断，第一次判断预读窗口是否处于关闭状态，

/* 同步预读一页页面到内存中。 */

mapping: 文件拥有者的`addresss_space`对象

ra: 包含此页面的文件`file_ra_state`描述符

filp: 文件对象

offset: 页面在文件内的偏移量

req_size: 完成当前读操作需要的页面数

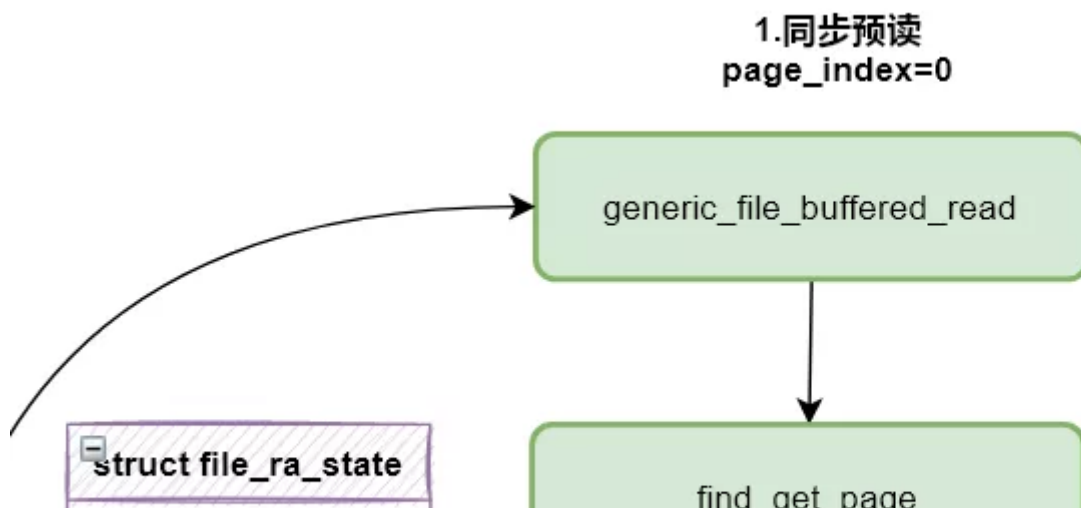
*/

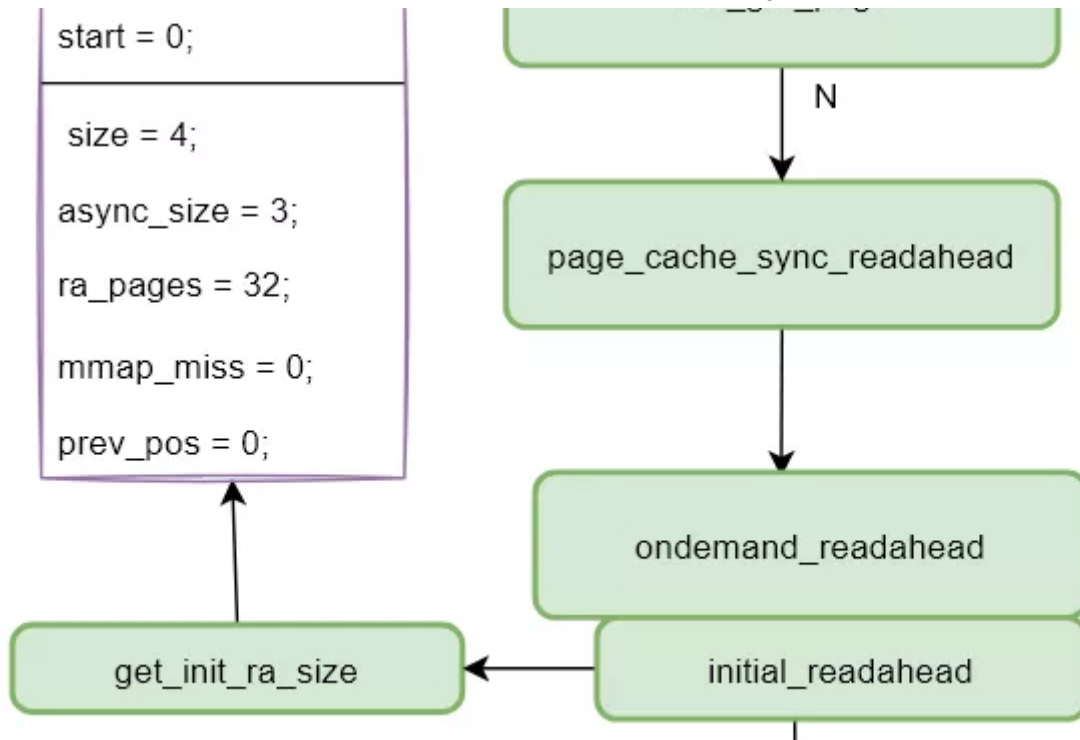
```
void page_cache_sync_readahead(struct address_space *mapping,
                               struct file_ra_state *ra, struct file *filp,
                               pgoff_t offset, unsigned long req_size)
{
    /* no read-ahead 预读窗口是否为关闭状态*/
    if (!ra->ra_pages)
        return;

    /* be dumb 当文件模式设置FMODE_RANDOM时, 表示文件预期为随机访问, 这种情形比较少见。*/
    if (filp && (filp->f_mode & FMODE_RANDOM)) {
        force_page_cache_readahead(mapping, filp, offset, req_size);
        return;
    }

    /* do read-ahead */
    ondemand_readahead(mapping, ra, filp, false, offset, req_size);
}
EXPORT_SYMBOL_GPL(page_cache_sync_readahead);
```

第二次判断文件是否被预设为随机访问，最后调用`ondemand_readahead`函数，该函数对同步预读窗口进行初始化。通过`get_init_ra_size`函数计算预读窗口长度（`ra->size`）。





计算窗口长度的函数get_init_ra_size如下：

```

static unsigned long get_init_ra_size(unsigned long size, unsigned long max)
{
    //size = reqsize = last_index-index;
    unsigned long newsize = roundup_pow_of_two(size); //四舍五入到最近的2次幂

    if (newsize <= max / 32) //读的小 <1
        newsize = newsize * 4; //预读四倍
    else if (newsize <= max / 4) //读的中等1-8
        newsize = newsize * 2; //预读2倍
    else //>8
        newsize = max;
    return newsize;
}
  
```

通过打印预读窗口变量*ra，可以看到初始化的窗口变量取值如下图所示。

```

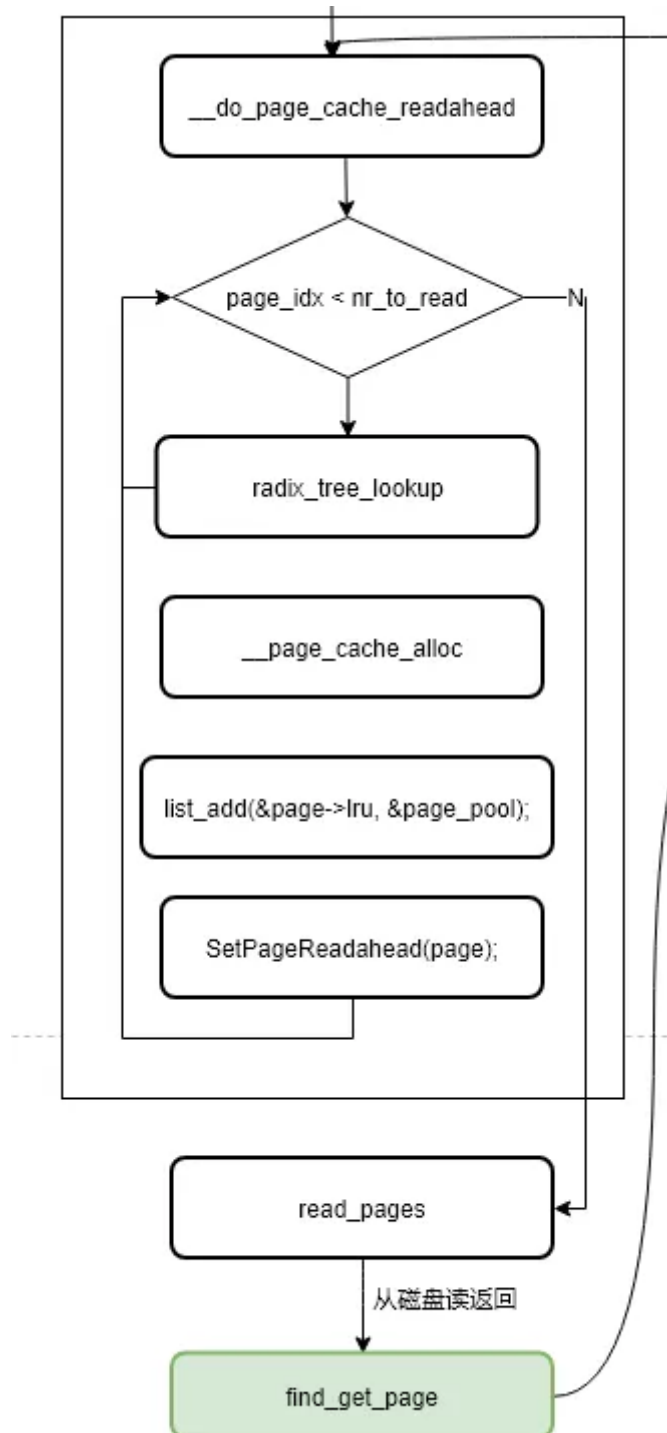
(gdb) p *ra
$10 = {
    start = 0,
    size = 4,
    async_size = 3,
    ra_pages = 32,
    mmap_miss = 0,
  }
  
```



```
prev_pos = 0
}
```

预读窗口

预读窗口取值确定以后，就该调用函数__do_page_cache_readahead进行页的分配与磁盘数据的读取，函数流程图如下图所示。



函数主体功能如下，循环执行 $ra \rightarrow size$ 次，即读取预读窗口中的每个page，调用__page_cache_alloc依次分配page，并保存在链表page_pool中，根据页索引（ $page_idx = size - async_size$ ）的取值给预读的页设置PageReadahead标识，当用户态程序读到该标识页时进行下

一次异步预读。最后调用read_pages进行磁盘读操作，完成后再次执行find_get_page，就能够从缓存中命中页面。

```
int __do_page_cache_readahead(struct address_space *mapping, struct file *filp,
    pgoff_t offset, unsigned long nr_to_read,
    unsigned long lookahead_size)
{
    ...
    LIST_HEAD(page_pool); // 将要读取的页存入到这个list当中
    ...
    end_index = ((isize - 1) >> PAGE_SHIFT);

    /*
     * Preallocate as many pages as we will need.
     再次检查页面是否已经被其他进程读进内存，如果没有则申请页面。
     nr_to_read是预读窗口的大小，ra->size。
     */
    for (page_idx = 0; page_idx < nr_to_read; page_idx++) {
        pgoff_t page_offset = offset + page_idx; // 计算得到page index

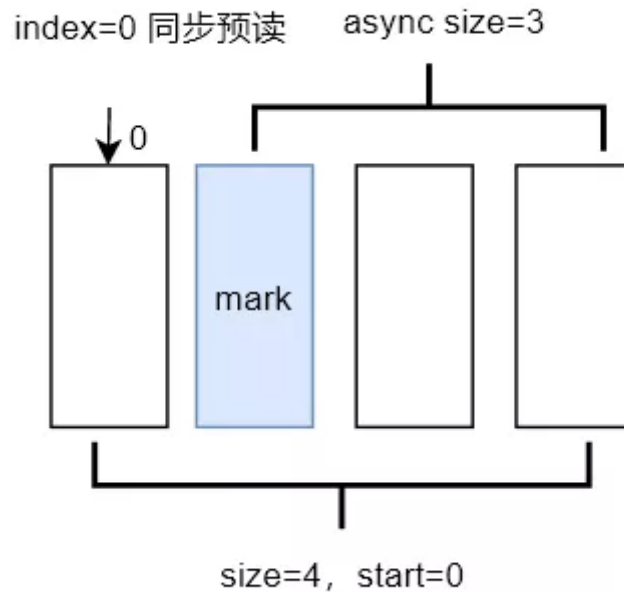
        if (page_offset > end_index) // 超过了文件的尺寸就break，停止读取
            break;

        rcu_read_lock();
        // 查看是否在page cache，如果已经在cache中，再判断是否为脏，要不要进行读取
        page = radix_tree_lookup(&mapping->page_tree, page_offset);
        rcu_read_unlock();
        if (page && !radix_tree_exceptional_entry(page))
            continue;
        // 如果不存在，则创建一个page cache结构
        page = __page_cache_alloc(gfp_mask);
        if (!page)
            break;
        // 设定page cache的index
        page->index = page_offset;

        // 加入到list当中
        list_add(&page->lru, &page_pool);
        // 当分配到第nr_to_read - lookahead_size个页面时，就设置该页面标志PG_readahead，以让下次进行异步预
        if (page_idx == nr_to_read - lookahead_size)
            SetPageReadahead(page);
        ret++;
    }
}
```

```
...  
  
if (ret)  
    read_pages(mapping, filp, &page_pool, ret, gfp_mask);  
...  
}
```

下图可以很好的展示首次同步预读后的预读窗口情况，以及当前缓存中数据的状况：



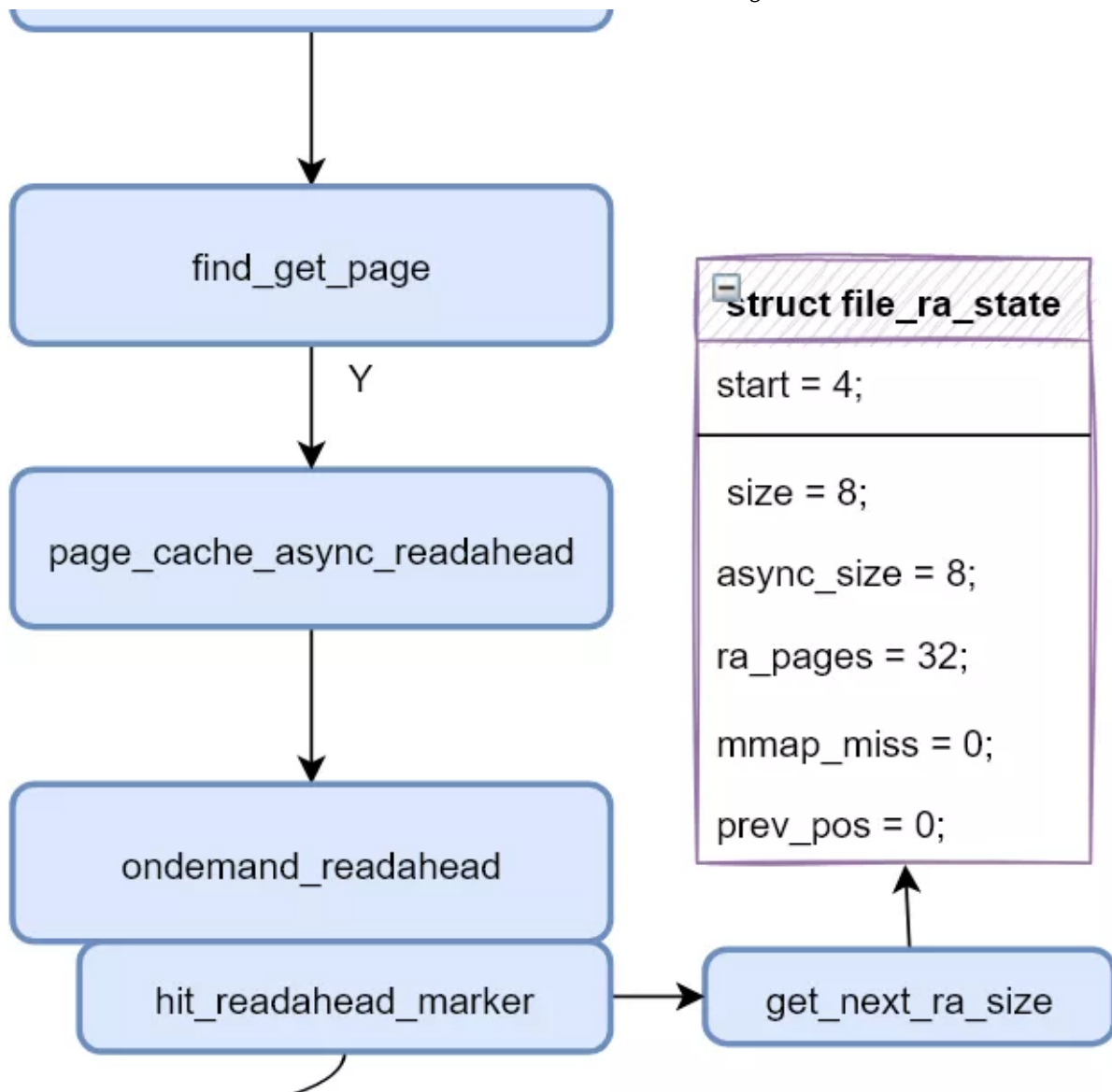
用户态当前正在读取的page index 为0，预读窗口size为4，预读4个页面到内存后，给页索引（ $\text{index} = \text{size} - \text{async_size}$ ）为1的页设置了预读标识（图中蓝色的page），该标识会在下一次用户态程序读到该page时触发异步预读。

后续异步预读

第一次读取完成后，用户态程序会进行第二次循环读取（`read(in, &c, 4096)`），因为程序设置了每次读取一页的大小4k，因此第二次刚好读取`page_index = 1`的页。`generic_file_buffered_read`同样还是率先进行`find_get_page`，期望能够从页缓存中获取到`index=1`的page。幸运的是，上次读取进行同步预读，预读了下标为0-3的4个页面，本次可以在缓存中命中页面。

2. 异步预读 page_index=1, 4

`generic_file_buffered_read`



命中缓存之后，会进行异步预读标识的判定，执行流程如上图所示，查看当前读取的页面（本次page_index=1）是否被预读窗口标识了预读标识PG_readahead。

```

if (PageReadahead(page)) { //检测页标志是否设置了PG_readahead, 启动异步预读
    page_cache_async_readahead(mapping,
        ra, filp, page,
        index, last_index - index);
}

```

在上次同步之后，标识了页索引（index）为1的page为PG_readahead，所以本次读取会触发异步预读。触发异步预读之后，page_cache_async_readahead函数会清除掉当前页面（page）的预读标志PG_readahed，然后调用ondemand_readahead函数。和上面的同步预读相似，在ondemand_readahead函数中会重新设置预读窗口长度（ra->size），通常会扩大为原来长度的2倍或者4倍。

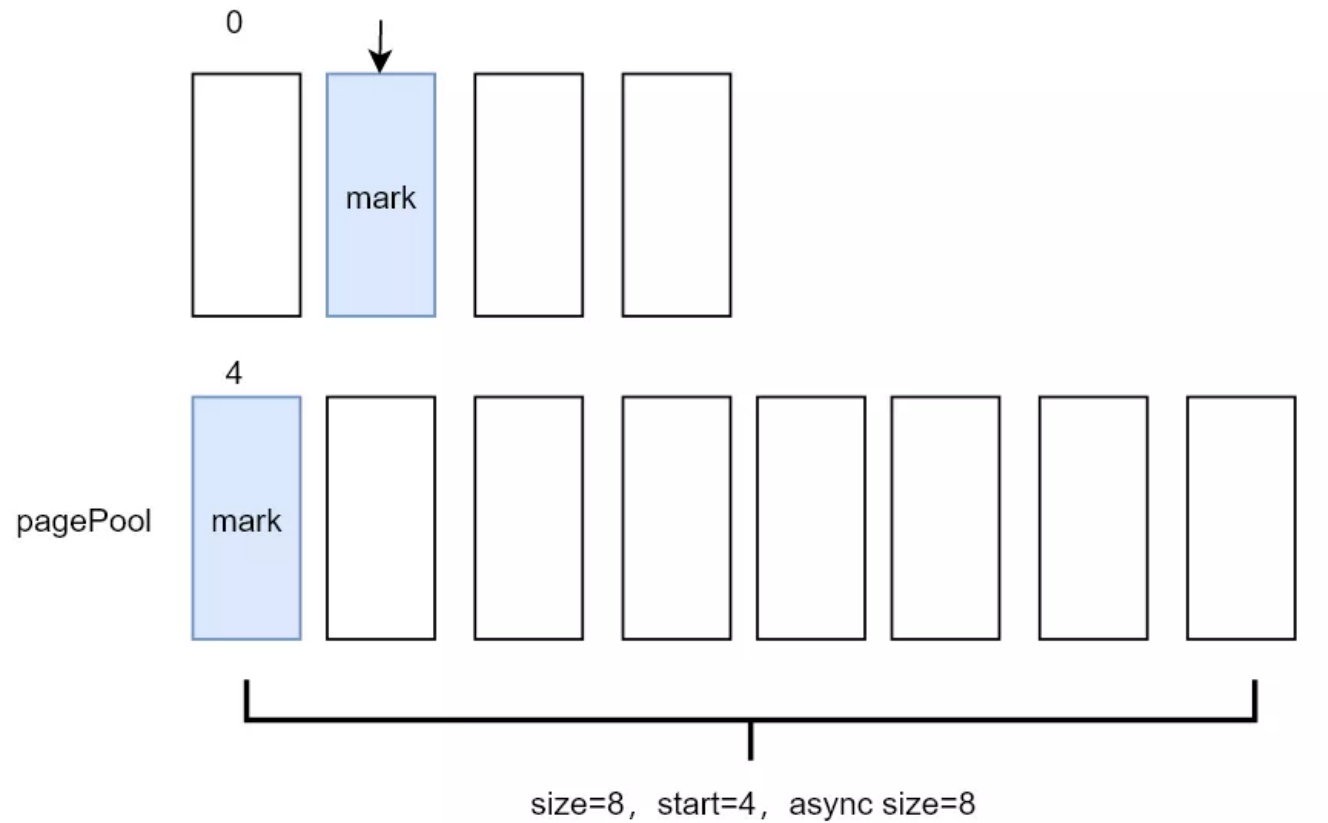
打印更新后的预读窗口变量，start为4，表示窗口往后推移了4个页面，将页索引为4的页面作为窗口开始；size为8，表示窗口的长度扩大为原来的二倍，因为算法觉察到用户态程序是在顺序读取，加大预读的页数；async为8，表示为当前窗口的第一个（index=0）页设置预读标识。

```
(gdb) p *ra
$29 = {
  start = 4,
  size = 8,
  async_size = 8,
  ra_pages = 32,
  mmap_miss = 0,
  prev_pos = 0
}
```

预读窗口

接下来的执行就与首次同步预读的流程基本一致了。异步预读后的窗口和当前内存缓存页的情况如下图所示，本次预读了页索引4-11共8个page，并给index=4的page设置了预读标识，当前用户态程序读取的page_index=1。

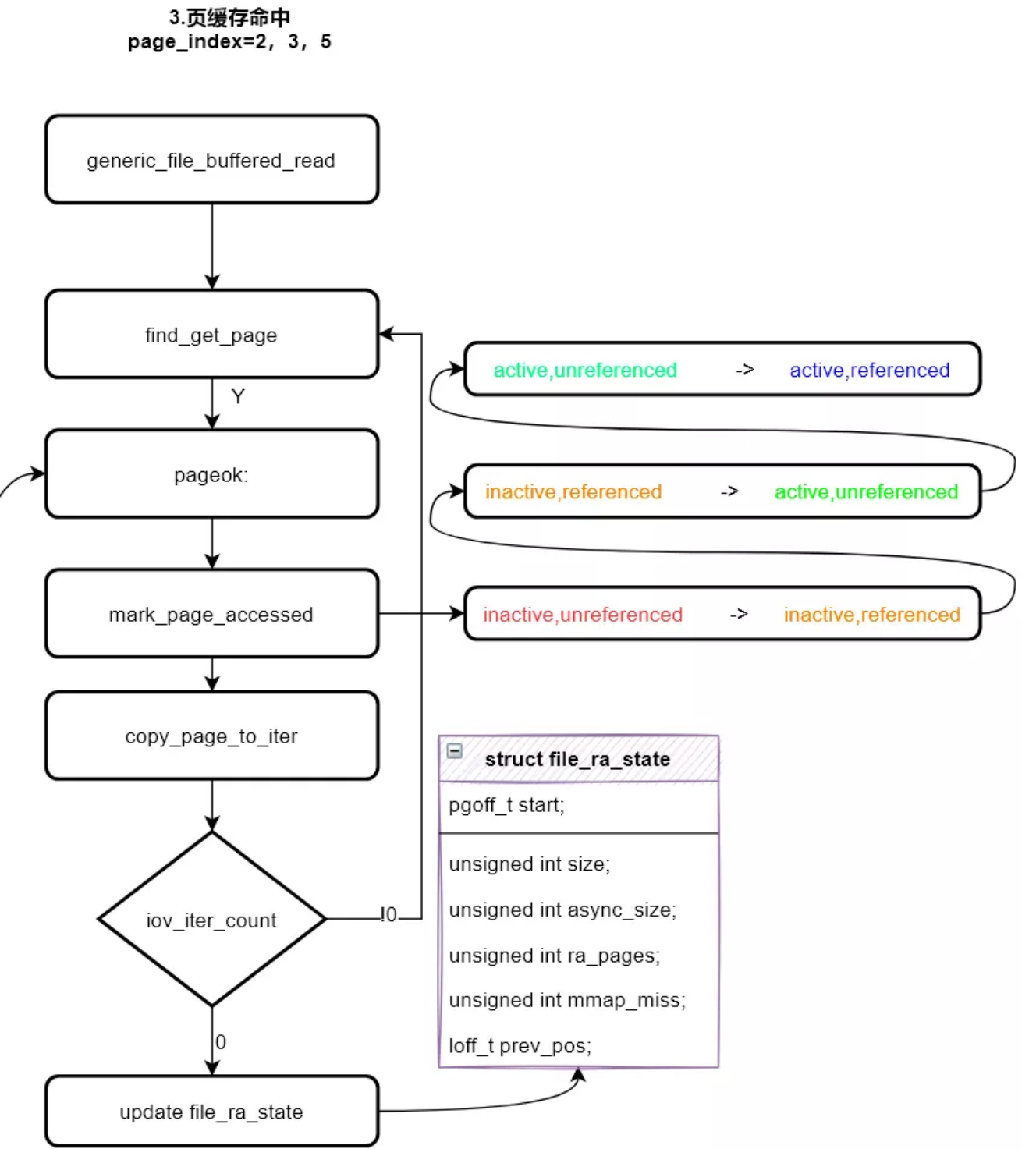
index=1 异步预读



缓存状态

后续缓存命中读取

由于之前的两次预读，该文件的前12个page已经缓存在内存中，第三次read会直接命中缓存，并且该页没有设置PG_readahead标识，也不会触发异步预读。



`find_get_page`函数命中缓存后，会执行`pageok`代码片段，然后调用`mark_page_accessed`函数，为page在其所属的LRU链表中提升等级。如上图所示，共分成四个等级，分别用四种颜色表示。`inactive,unreferenced` 为最不活跃页面，`active,referenced`为最活跃页面。

inactive, referenced	->	active, unreferenced
active, unreferenced	->	active, referenced

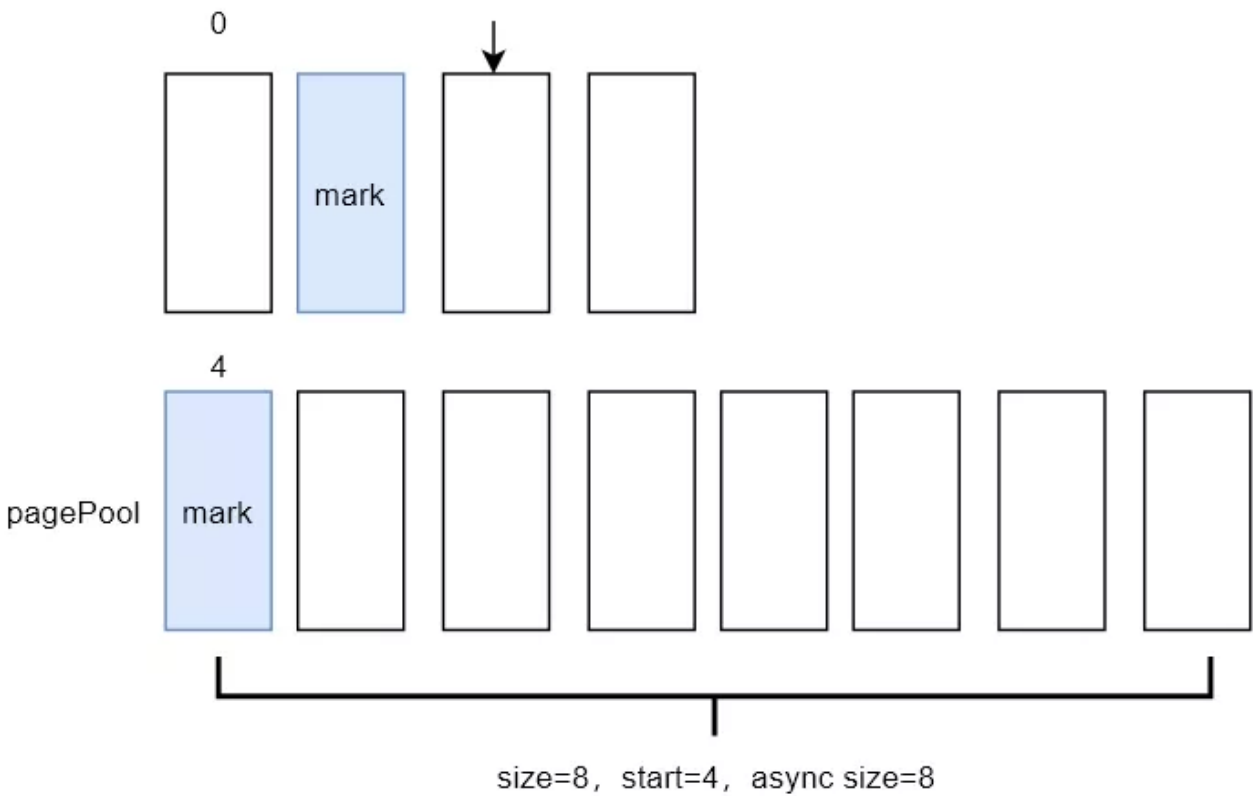
本次读取完成后，可以看到预读窗口的大小没有发生变化，只更新了prev_pos，表示上次读文件的起始偏移位置。

```
(gdb) p *ra
$43 = {
  start = 4,
  size = 8,
  async_size = 8,
  ra_pages = 32,
  mmap_miss = 0,
  prev_pos = 8192
}
```

预读窗口

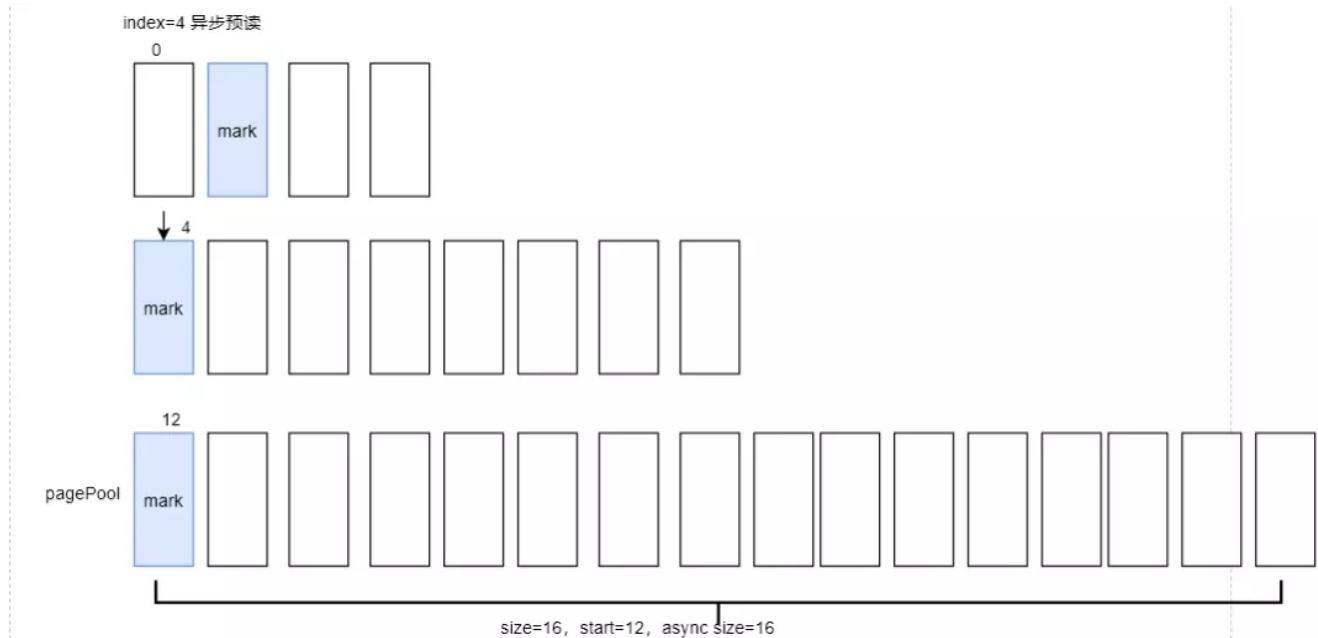
读取page_index=2,3的页面都会缓存命中且不会预读，窗口如下。我们让用户态程序快进到读取index=4的page，内核会再次启动预读。

index=2,3 缓存命中



读取页索引（page_index）为4的页面

page_index=4的页面被标识了PG_readahead，因此会再次启动异步预读，窗口大小变为原来的2倍，即size=16。本次预读了index=12-27，共16个页面，并且index=12的页面会被标识PG_readahead。



GDB打印窗口变量如下：

```
(gdb) p *ra
$53 = {
  start = 12,
  size = 16,
  async_size = 16,
  ra_pages = 32,
  mmap_miss = 0,
  prev_pos = 16384
}
```

预读窗口

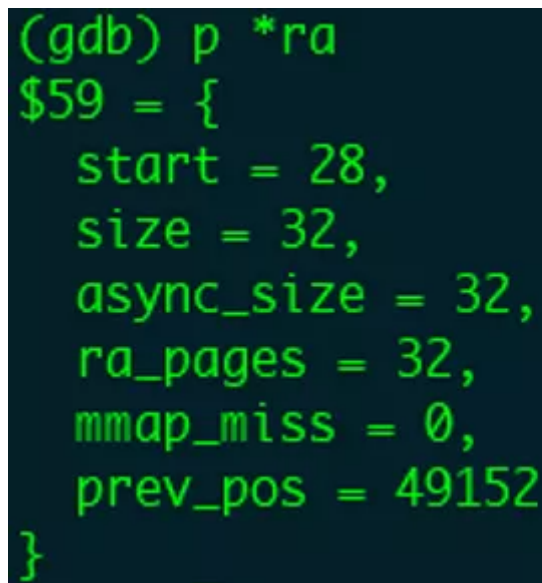
当前用户态程序读取的page_index=4，后面的页面5-11会全部命中缓存，并且不会预读，我们直接跳到用户态程序读取page_index=12的页面时，这时还会触发异步预读。

读取页索引（page_index）为12的页面

本次预读窗口的size会变成32，也就是会预读32个后续页面，但是由于news.txt总共包含32个页面，上一次已经预读了28个页面index=0-27，本文件还剩余4个页面。在__do_page_cache_readahead函数中实际分配预读的page时，内核会通过文件的大小，计算文件的最大page_index。同时在分配页面的时候进行判断，一旦超过了end_index，就会终止页面的分配。

```
end_index = ((isize - 1) >> PAGE_SHIFT);  
...  
pgoff_t page_offset = offset + page_idx; // 计算得到page index  
...  
if (page_offset > end_index) // 超过了文件的尺寸就break，停止读取  
    break;
```

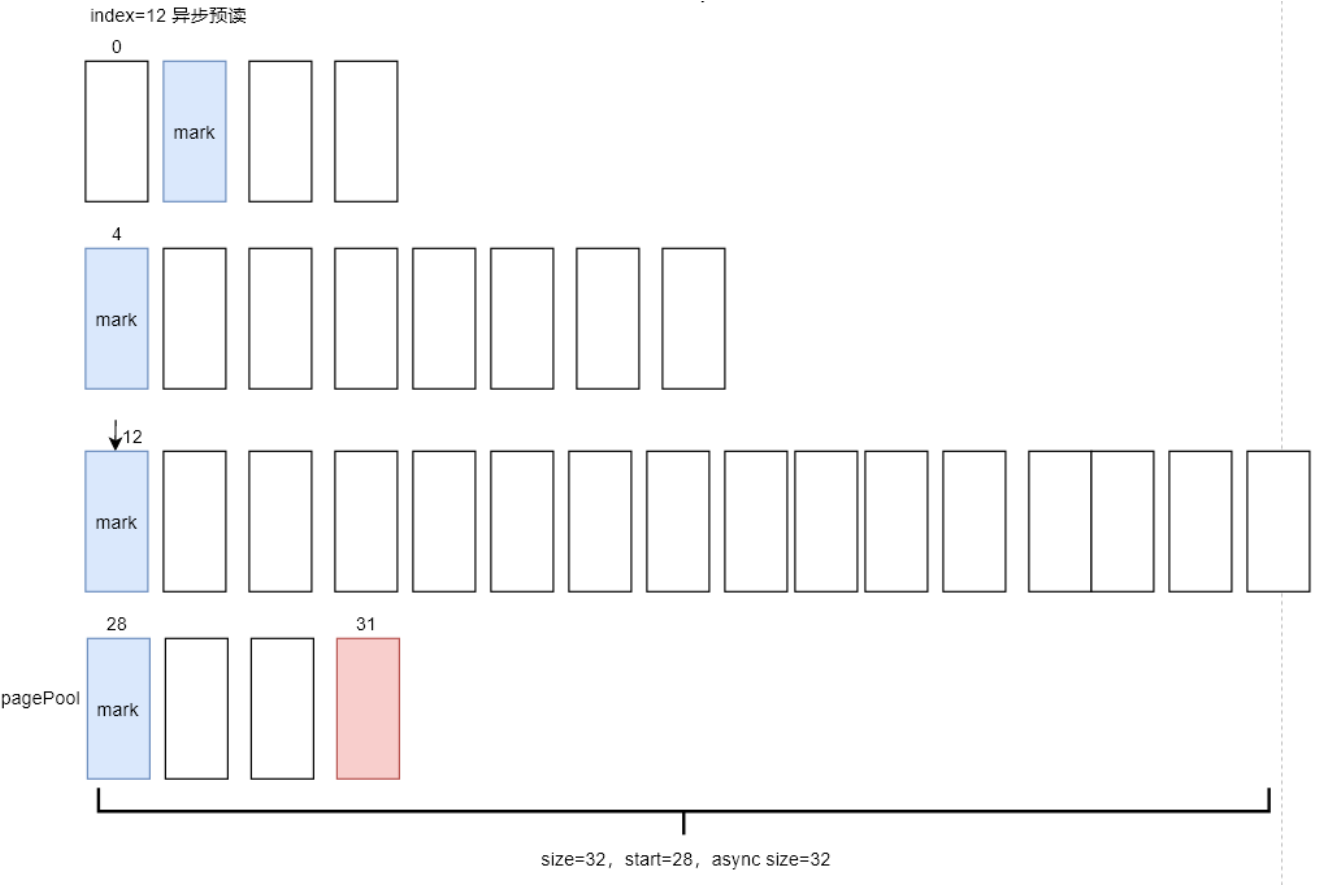
GDB打印窗口变量如下：



```
(gdb) p *ra  
$59 = {  
    start = 28,  
    size = 32,  
    async_size = 32,  
    ra_pages = 32,  
    mmap_miss = 0,  
    prev_pos = 49152  
}
```

预读窗口

因此本次异步预读窗口size虽然是32，但是由于目标文件只有4个剩余page，最终也只会分配4个page进行数据的缓存。最后还会给page_index=28的页面设置PG_readahead标识，当读到该页面时还是会触发异步预读。



读取页索引（page_index）为28的页面

本次读取窗口size依然会更新，但是窗口size不能大于最大值ra_pages=32，因此本次预读还是32个page，同样因为文件已经没有数据，并不会进行实际的page分配和磁盘读取。

读取page_index=28的页面后的窗口：

```
(gdb) p *ra
$69 = {
  start = 60,
  size = 32,
  async_size = 32,
  ra_pages = 32,
  mmap_miss = 0,
  prev_pos = 114688
}
```

预读窗口

读取page_index=31的页面后的窗口：

```
(gdb) p *ra
$78 = {
  start = 60,
  size = 32,
  async_size = 32,
  ra_pages = 32,
  mmap_miss = 0,
  prev_pos = 130596
}
```

预读窗口

至此整个news.txt文件全部读取完成，数据也都缓存在了内存中。后续将进行更为复杂的文件读取模式流程分析！

参考内容：

[1]吴峰光. Linux内核中的预取算法[D].中国科学技术大学,2008.

 期待与你共同成长



知书码迹

融实用性和趣味性于一体，专注于Linux内核、JAVA、数据结构算法面试题、物联网网络...

18篇原创内容

公众号



微信号：知书码迹

更多精彩内容：<https://szp2016.github.io/>

喜欢此内容的人还喜欢

Linux 环境变量解读

我的Python之人工攻智能路

Linux目录结构详解

https://mp.weixin.qq.com/s/VPje2PX56iovJ9VprfEz_g

DevHome

实用20个linux小技巧，总有你没有用过的！
浩道linux