

深入理解Linux内核共享内存机制- shmem&tmpfs

原创

OPPO内核工匠

于 2024-02-02 17:31:43 发布

CC 4.0 BY-SA版权

阅读量3.4k

收藏 31


点赞数 30

文章标签：

linux

运维

服务器

 腾讯云开发者社区

文章已被社区收录

加入社区

搞过Linux大家都知道，Linux的世界中，进程的虚拟地址空间有两部分组成：内核空间和用户空间，内核空间各个进程直接共享，而用户空间彼此隔离，大家井水不犯河水。但是并不是老死不相往来，我们有时候需要进程直接共享一些数据，于是乎，Linux就有了共享内存的机制。

我们在使用Linux时，匿名页和文件页这两种类型的页面经常在我们耳边回荡，我们或多或少都知道，文件页会关联文件系统中的文件，而匿名页不关联任何文件，但是经常在回收时会将其保存到交换设备（前提是系统打开了交换设备）。你是否知道，Linux世界中还存在第三种页面-共享内存页，它是那么的特殊，以至于同时具备文件页和匿名页的一些特征（如会关联文件，存在page cache，同时也具备交换功能），正所谓是“跨界的老演员”了。

本文将揭开 **Linux共享内存** 的神秘面纱，来看看共享内存页是如何跨越“跨越两界”，如何实现了跨进程共享内存，又是如何被系统回收的。

1.应用场景

目前linux系统中，shmem的应用场景如下：

场景	说明	关键调用链
共享匿名映射	父子进程间通信	//mm/mmap.c mmap_region ->shmem_zero_setup
ipc共享内存	任意进程间共享内存	//ipc/shm.c newseg ->shmem_kernel_file_setup ->__shmem_file_setup
tmpfs	实现内存文件系统	//mm/shmem.c shmem_file_operations.mmap ->shmem_mmap ->vma->vm_ops = &shmem_vm_ops
ashmem	安卓匿名共享内存	//drivers/staging/android/ashmem.c ashmem_mmap ->shmem_file_setup ->__shmem_file_setup ->alloc_file_pseudo(inode, mnt, name, O_RDWR, &shmem_file_operations)
memfd	创建共享匿名文件, 后面主要讲解memfd，目前新的内核版本memfd已经替代了安卓中大名鼎鼎的ashmem。	//mm/memfd.c memfd_create ->shmem_file_setup
其 他 ： 如 gpu (kgsi)、/dev/zero等	N/A	N/A

下面整理出常用的shmem的接口：

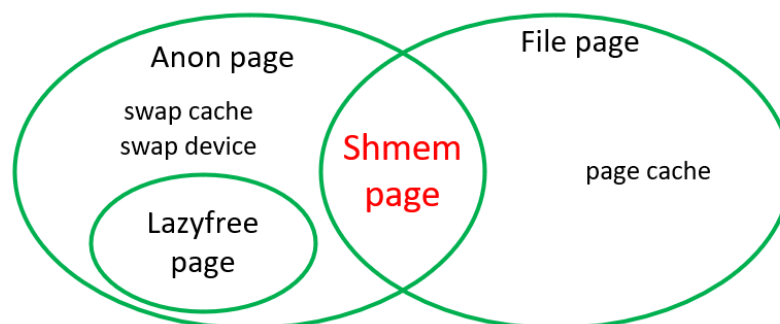
接口	说明
----	----

shmem_kernel_file_setup	使用内部的shm_mnt，获得一个关联了shmem的file，用于内核内部使用
shmem_file_setup	使用内部的shm_mnt，获得一个关联了shmem的file
shmem_file_setup_with_mnt	使用传递的mnt，获得一个关联了shmem的file
shmem_zero_setup	设置共享匿名映射
shmem_read_mapping_page_gfp	按照 page cache -> swap cache -> swap device顺序查找页面，没有则分配并加入page cache
reclaim_shmem_address_space	回收mapping的页面

2.页面类型和特点及共享原理

2.1 页面类型

通常Linux系统中，主要的页面类型如下：



而shmem页面既有匿名页的特点（page->flags设置PG_swapbacked，具有swap功能），也有文件页的特点（inode->i_mapping->a_ops = &shmem_aops，关联文件，有page cache）。

2.2 LRU类型

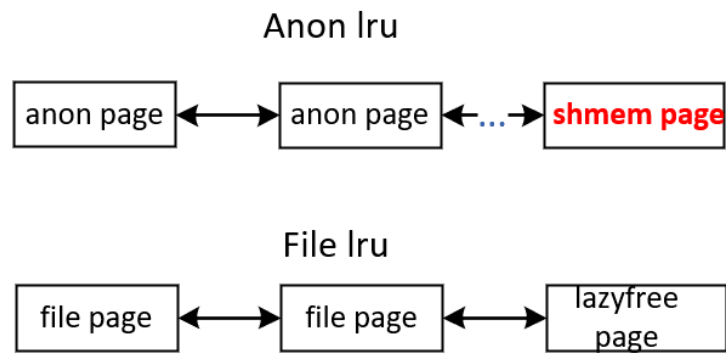
通常Linux中，用户态进程使用的物理页面会放入LRU链表中进行老化/回收，匿名页面会加入匿名lru，而文件页会加入文件lru。

内核源码中判断是否为文件lru的注释如下：

```

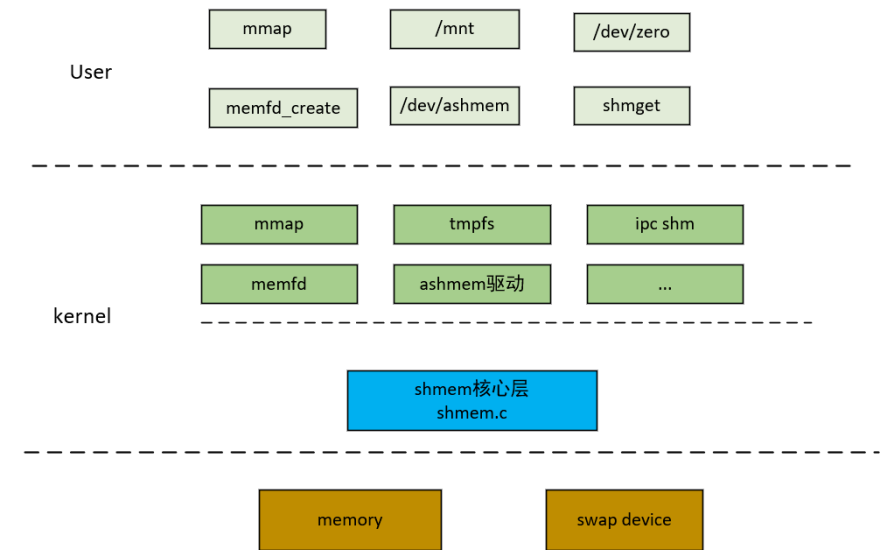
/**
 * folio_is_file_lru - Should the folio be on a file LRU or anon LRU?
 * @folio: The folio to test.
 *
 * We would like to get this info without a page flag, but the state
 * needs to survive until the folio is last deleted from the LRU, which
 * could be as far down as __page_cache_release.
 *
 * Return: An integer (not a boolean!) used to sort a folio onto the
 * right LRU list and to account folios correctly.
 * 1 if @folio is a regular filesystem backed page cache folio
 * or a lazily freed anonymous folio (e.g. via MADV_FREE)
 * 0 if @folio is a normal anonymous folio, a tmpfs folio or otherwise
 * ram or swap backed folio.
 */
static inline int folio_is_file_lru(struct folio *folio)
{
    return !folio_test_swapbacked(folio);
}
  
```

虽然shmem页面既有匿名页特定又有文件页特点，但是由于它有swap特性，它会加入到匿名的lru中。



2.3 shmem框架

下面给出shmem的整体框架：



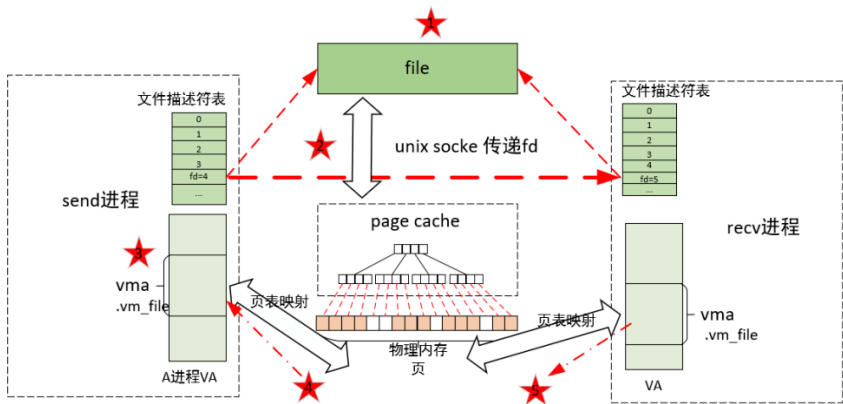
各层的作用说明如下表：

软件层	作用	说明
用户接口层	实现了用户的接口，对用户可见。	1.通过文件系统节点暴露给用户空间（如： <code>/dev/ashmem</code> 、 <code>tmpfs</code> 等）。 2.通过系统调用接口供用户使用（如： <code>memfd_create</code> 、 <code>mmap</code> 系统调用）。
shmem使用者	使用shmem框架来实现自己的功能。	如 <code>memfd_create</code> 创建一个匿名文件， <code>mmap</code> 利用shmem实现共享匿名映射， <code>tmpfs</code> 利用shmem实现内存文件系统等
shmem核心层	提供内存分配、回收等核心功能。	主要在 <code>mm/shmem.c</code> 文件中

注：这里仅仅是为了理解方便，内核中并没有这样严格的划分！

2.4 内存共享原理

内存共享原理框图如下：



如图所示，以memfd为例，实现内存共享的步骤如下：

1) 通过memfd系统调用等方式创建文件描述符（fd）。

例子中send进程会通过memfd系统调用来获得一个没有使用的fd，并将fd关联文件实例（file），这个file就会关联一片共享内存。

2) 将文件描述符传递给其他进程来实现共享。

如例子中通过unix socket传递文件描述符，实际上传递文件描述符是在接收方申请一个没有使用的文件描述符，然后关联共享内存对应的file。

例如：send进程的文件描述符fd=4会关联共享内存对应的file，recv进程的文件描述符fd=5也会关联共享内存对应的file。

可以看的出来，虽然是传递文件描述符，但是他们的相关文件描述符并不一定一样，只是指向相同的关联共享内存的file罢了。

3) send/recv进程通过mmap映射共享内存到进程虚拟地址空间。

4) send进程首次写访问数据

这个时候会发生缺页异常，page cache查询不到物理页面PAGE1，会申请物理页面并加入文件实例对应的page cache，并通过页表映射PAGE1到send进程的虚拟地址空间。

缺页返回后，将数据写入PAGE1（这里都是'a'）。

5) recv进程首次读访问数据

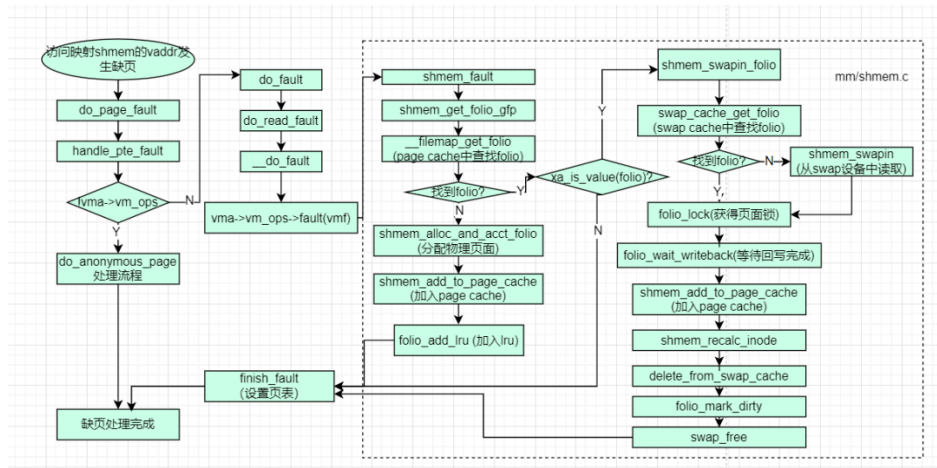
同样也会发生缺页异常，但是会首先查询page cache，发现PAGE1，然后通过页表映射PAGE1到recv进程的虚拟地址空间。

缺页返回后，从PAGE1读出数据（这里都是'a'），于是实现了内存共享。

注：通过memfd来共享内存的实践见最后一章讲解。

3.缺页处理

缺页处理流程如下：



缺页处理步骤框图如下：



下面讲解下，shmem页面的缺页处理步骤：

缺页发生时，

1) 查找或分配物理页面

1.1 先从page cache中查找

相关的物理页面可能已经被其他线程加入了page cache，所以首先从page cache查找。

1.2 找不到从swap cache中查找

页面有可能在回收等场景被加入了swap cache，所以在这里也查找下。

1.3 找不到如果之前有swap out 则swap in

之前如果由于内存回收等场景相关页面被swap out到swap device,那么相关的swap cache对应的位置会被替换为swap entry, 这个时候根据swap entry从swap device中读取物理页面内容。

1.4 否则分配新的folio

上面都尝试了查找但是没有找到，那么有可能是第一次访问这个页面，这个时候需要分配新的物理页面，既是folio。

2) 加入lru

shmem会被加入匿名的lru中，以便内存回收都场景回收到swap device。

3) 页表映射

将相关的物理页面通过页表映射到进程的虚拟地址空间，这样后面进程就可以正常访问页面数据了。

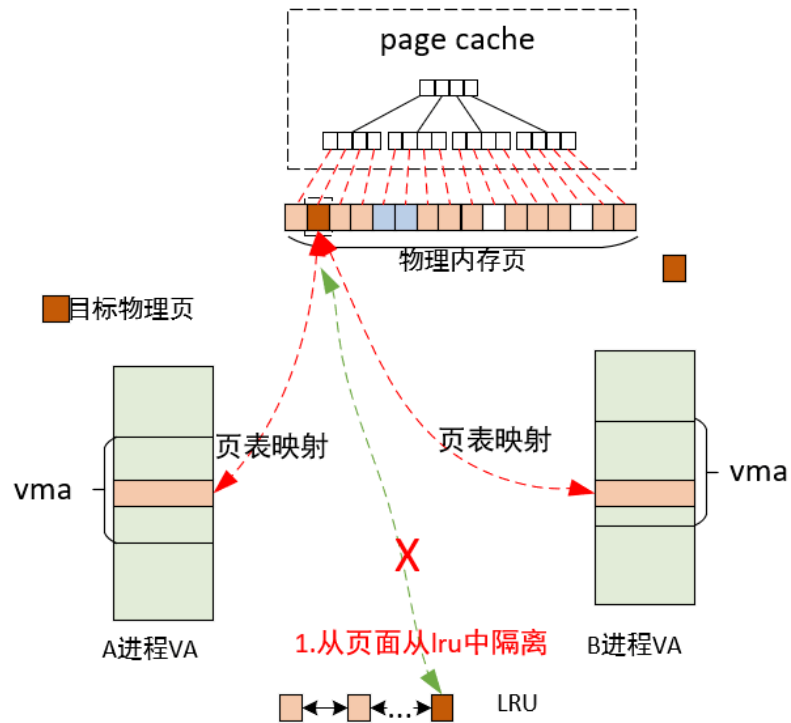
4.回收shmem页

回收shmem页面流程如下：

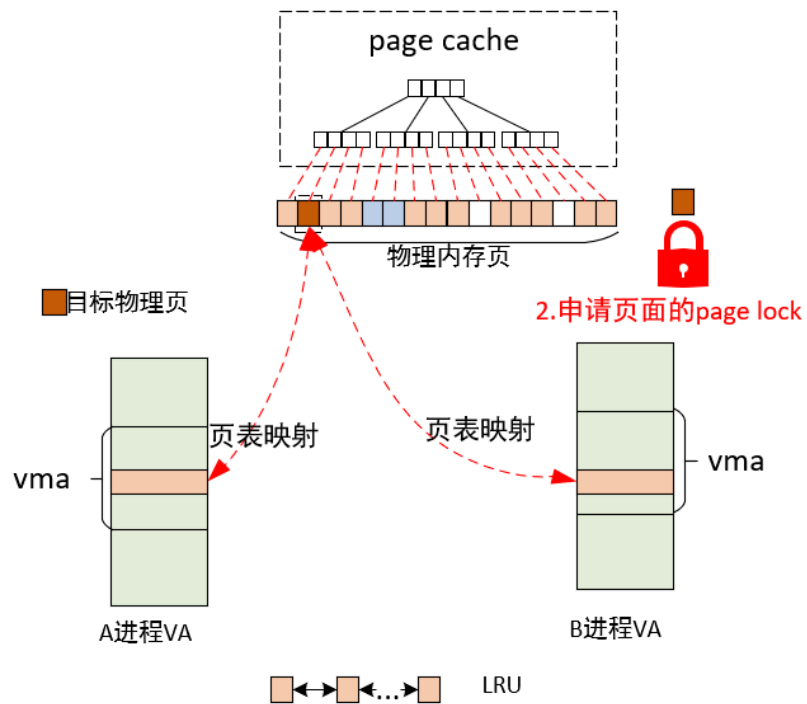


由于shmem页面回收比较复杂，下面我们分主要步骤进行图解：

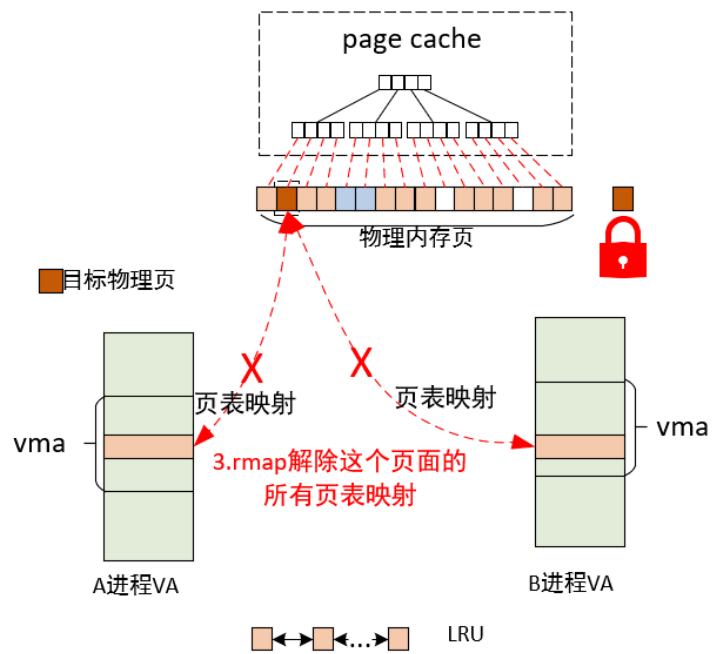
1) 从页面从lru中隔离



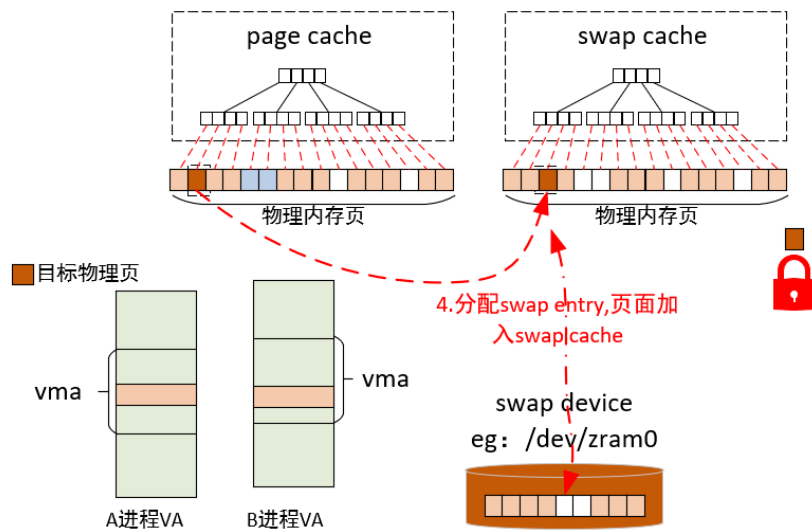
2) 申请页面的page lock



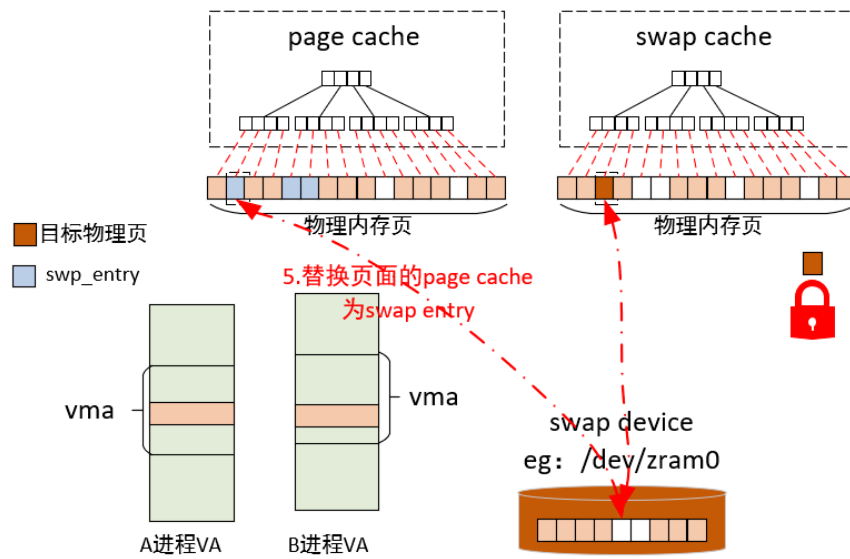
2) rmap解除这个页面的所有页表映射



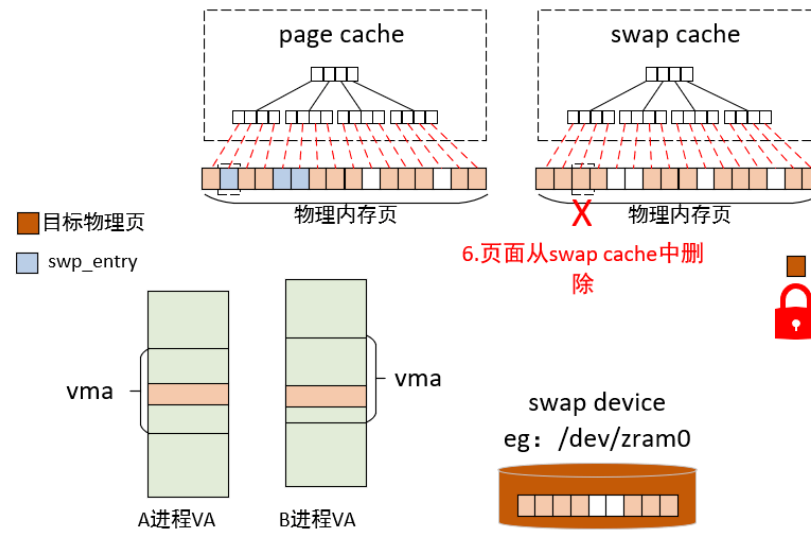
4) 分配swap entry,页面加入swap cache



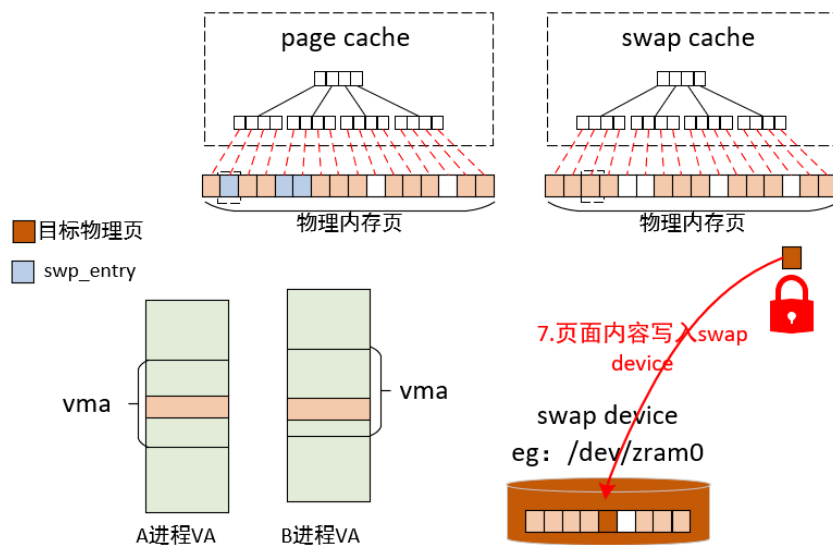
5) 替换页面的page cache为swap entry



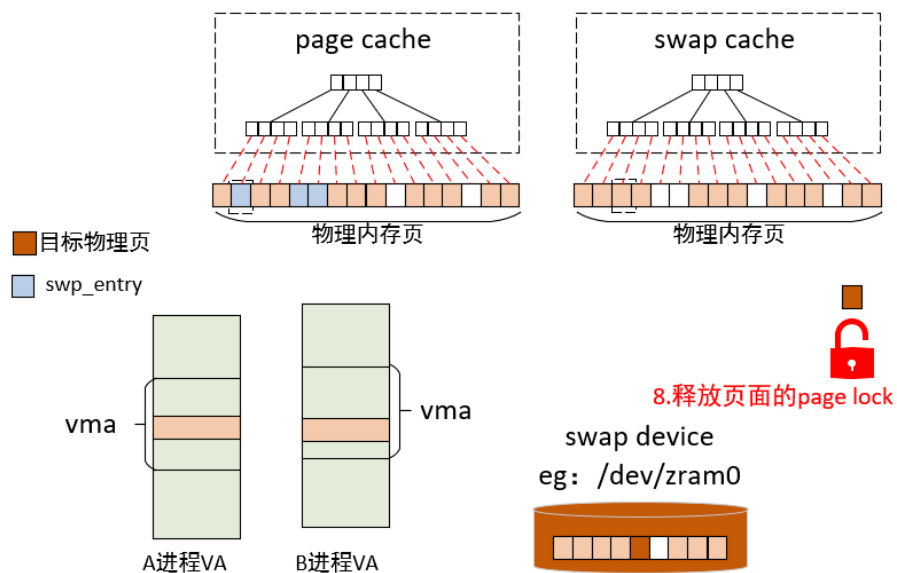
6) 页面从swap cache中删除



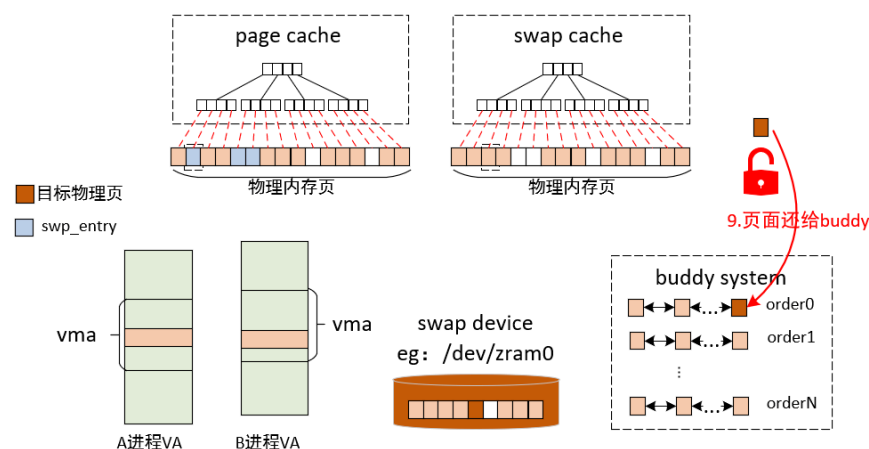
7) 页面内容写入swap device



8) 释放页面的page lock



9) 页面还给buddy



这里需要注意一点的是：shmem页面回收时保存swap entry的方式跟匿名页完全不一样，匿名页在回收时，会将相应的swap entry替换为原来的页表项，而shmem页面会直接清掉原来的页表项，会将swap entry替换为对应的swap cache的位置。

5.tmpfs

Linux系统中，有一种文件系统叫做tmpfs，他的文件数据都在内存中，掉电会丢失，所以也有“临时文件系统”之意。

它有以下主要特点：

为内存文件系统，所有的文件数据都在内存中，掉电丢失。

由于数据在内存，数据访问速度很快。

内存不足，回收到swap中（如zram）。

读的时候，不分配物理页面，读取的数据都是0。

5.1读文件

读文件流程如下：



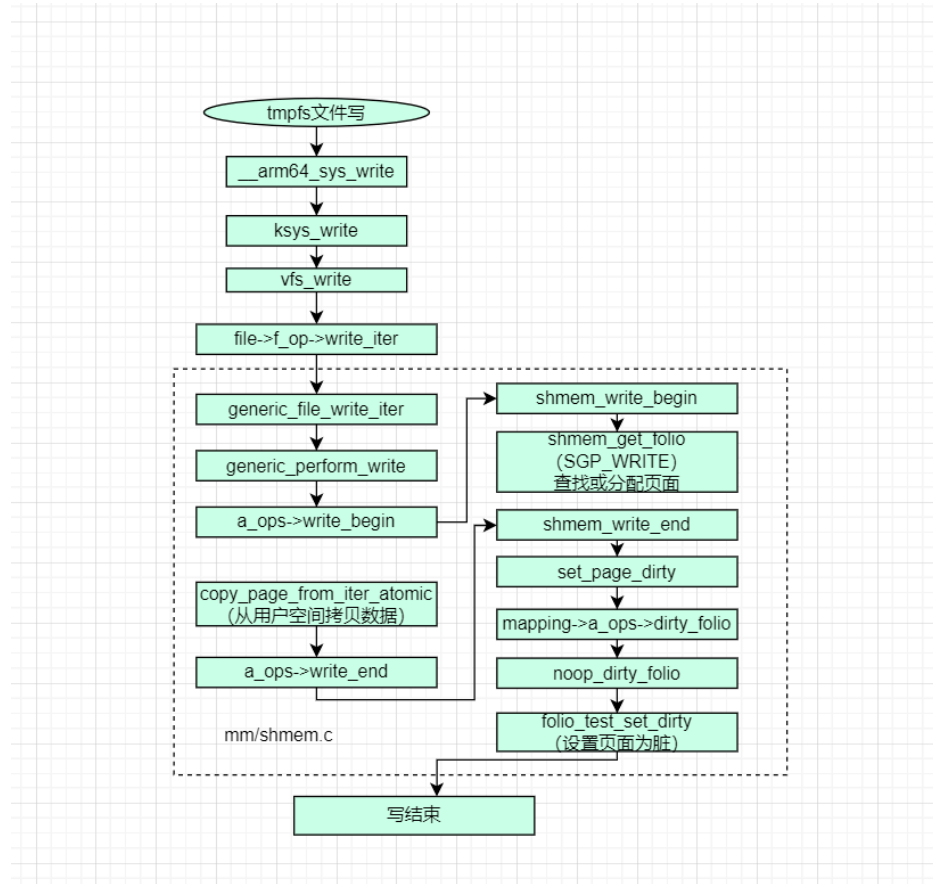
读tmpfs文件的主要步骤如下：

- 1) 按照page cache -> swap cache -> swap device顺序查找文件页面
- 2) 查找并拷贝页面内容到用户空间缓冲区
 - 如果找到，则拷贝文件页面数据到用户空间缓冲区。
 - 如果没有找到，则直接往用户空间缓冲区拷贝0。
- 3) 更新文件读写位置

这里需要注意的是：对于tmpfs文件系统中文件的读操作来说，按照page cache -> swap cache -> swap device顺序如果查找不到页面，则不会分配新的页面，只会往用户空间缓冲区拷贝0（这有点类似匿名页的第一次读，一般会映射到0页），这种情况也说明了相关文件偏移的页面从来没有被人写访问过。

5.2 写文件

写文件流程如下：



写tmpfs文件的主要步骤如下：

1) 查找或分配文件页面

- 按照page cache -> swap cache -> swap device顺序查找，如果找到，继续下一步
- 如果没找到，则分配新的页面

2) 从用户空间缓冲区拷贝数据到文件页面

3) 标记页面为脏

6.实践：通过memfd共享内存

下面给出完整测试case：

send.c

21399593791fe5268f861cbb08c64e36.png

```

37
38
39     socket_msg.msg_name = NULL;
40     socket_msg.msg_namelen = 0;
41     iov[0].iov_base = buf;
42     iov[0].iov_len = sizeof(buf);
43     socket_msg.msg_iov = iov;
44     socket_msg.msg_iovlen = 1;
45     socket_msg.msg_control = ctrl_data;
46     socket_msg.msg_controllen = sizeof(ctrl_data);
47
48     ctrl_msg = CMSG_FIRSTHDR(&socket_msg);
49     ctrl_msg->cmsg_len = CMSG_LEN(sizeof(sock_fd));
50     ctrl_msg->cmsg_level = SOL_SOCKET;
51     ctrl_msg->cmsg_type = SCM_RIGHTS;
52
53     /* send dmabuf_fd */
54     *((int *)CMSG_DATA(ctrl_msg)) = fd;
55
56     ret = sendmsg(sock_fd, &socket_msg, 0);
57     if(ret < 0) {
58         perror("client: sendmsg failed!\n");
59         return ret;
60     }
61 }
62
63 int main(int argc, char **argv)
64 {
65     int sfd, fd;
66     struct sockaddr_un addr;
67     char *ptr;
68
69     sfd = socket(AF_UNIX, SOCK_STREAM, 0);
70     if (sfd == -1)
71         errExit("fail to socket");
72
73     memset(&addr, 0, sizeof(struct sockaddr_un));
74     addr.sun_family = AF_UNIX;
75     strcpy(addr.sun_path, socket_file);
76

```

```

77     strcpy(addr.sun_path, socket_file);
78
79     #if 0
80     /* warning: implicit declaration of function 'memfd_create' */
81     fd = memfd_create(MEMFD_NAME, 0);
82 #else
83     fd = syscall(SYS_memfd_create, MEMFD_NAME, 0);
84 #endif
85     if (fd < 0)
86         errExit("fail to memfd_create");
87     else
88         debug("@ fd:%d @\n", fd);
89
90     ftruncate(fd, MAP_SIZE);
91
92     ptr = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
93     if (ptr == MAP_FAILED)
94         errExit("fail to mmap");
95
96     memset(ptr, 'a', MAP_SIZE);
97
98     if (connect(sfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_un)) == -1)
99         errExit("fail to connect");
100
101     send_fd(sfd, fd);
102     debug("send_fd ok!\n");
103
104     pause();
105     return 0;
106 }

```

recv.c

```

1 #define pr_fmt(fmt) "[recv]:" fmt
2
3 #define _gnu_source
4 #include <unistd.h>
5 #include <sys/syscall.h>
6 #include <linux/memfd.h>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <fcntl.h>
11 #include <sys/types.h>
12 #include <sys/socket.h>
13 #include <sys/mman.h>
14 #include <sys/un.h>
15 #include <sys/stat.h>
16 // #include <sys/memfd.h>
17
18 #define errExit(msg) do { perror(msg); exit(EXIT_FAILURE); \
19                     } while(0)
20
21 #define debug(fmt, ...) \
22     printf(pr_fmt(fmt), ##__VA_ARGS__);
23
24 #define MAP_SIZE 0x1000
25 #define MEMFD_NAME "my_memfd"
26
27 const char socket_file[] = "my_socket";
28
29 int recv_fd(int sock_fd)
30 {
31     int ret = -1;
32     struct msghdr socket_msg;
33     struct cmsghdr *ctrl_msg;
34     struct iovec iov[1];
35     socklen_t cli_len;
36     char buf[100];
37     char ctrl_data[MSG_SPACE(sizeof(sock_fd))];
38     int fd;

```

```

39
40     socket_msg.msg_name = NULL;
41     socket_msg.msg_namelen = 0;
42     iov[0].iov_base = buf;
43     iov[0].iov_len = sizeof(buf);
44     socket_msg.msg_iov = iov;
45     socket_msg.msg_iovlen = 1;
46     socket_msg.msg_control = ctrl_data;
47     socket_msg.msg_controllen = sizeof(ctrl_data);
48
49     ret = recvmsg(sock_fd, &socket_msg, 0);
50     if(ret < 0) {
51         printf("server: recvmsg failed!\n");
52         return ret;
53     }
54
55     ctrl_msg = CMSG_FIRSTHDR(&socket_msg);
56     if(ctrl_msg != NULL && ctrl_msg->cmsg_len == CMSG_LEN(sizeof(sock_fd))) {
57         if(ctrl_msg->cmsg_level != SOL_SOCKET) {
58             printf("cmsg_level is not SOL_SOCKET\n");
59             return ret;
60         }
61         if(ctrl_msg->cmsg_type != SCM_RIGHTS) {
62             printf("cmsg_type is not SCM_RIGHTS");
63             return ret;
64         }
65
66         fd = *((int *)CMSG_DATA(ctrl_msg));
67         debug("result: recv fd = %d\n", fd);
68
69         return fd;
70     }
71
72     return ret;
73 }

```

```

105
106     ptr = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
107     if (ptr == MAP_FAILED)
108         errExit("fail to mmap");
109
110     debug("recv data: \n");
111     for (i = 0; i < MAP_SIZE; i++)
112         printf("%c ", ptr[i]);
113
114     pause();
115     return 0;
116

```

send.c通过memfd_create系统调用创建了共享匿名文件，然后ftruncate设置文件大小，通过mmap将**共享内存映射**到自己的地址空间，再往这块内存种写全'a'，最后通过unix socket将文件描述符传递给recv进程。

recv.c通过unix socket接收send传递过来的文件描述符，通过mmap将共享内存映射到自己的地址空间，最后打印出这块共享内存内容。

执行结果如下：

```
[root@cheetah mnt]# ./recv &
[root@cheetah mnt]#
[root@cheetah mnt]#
[root@cheetah mnt]#
[root@cheetah mnt]#
[root@cheetah mnt]# ./send &
[root@cheetah mnt]# [send]: @ fd:4 @
[send]: send_fd ok!
[recv]:result: recv fd = 5
[recv]:@ fd:5 @
[recv]:recv data:
```

The terminal output shows a successful data transfer from a sender process to a receiver process. The receiver prints "recv data:" followed by a large grid of small square icons, each containing a single character. This visual representation indicates that the data has been successfully received and processed.

参考链接：

<https://elixir.bootlin.com/linux/v6.1.25/source/>

<https://elixir.bootlin.com/linux/v6.1.25/source/mm/shmem.c>

往

期

推

荐

Android分区挂载原理介绍（上）

Android分区挂载原理介绍（下）

指纹驱动初始化引发的思考