Hugetlbfs Reservation

Overview

Huge pages as described at :ref:`hugetlbpage` are typically preallocated for application use. These huge pages are instantiated in a task's address space at page fault time if the VMA indicates huge pages are to be used. If no huge page exists at page fault time, the task is sent a SIGBUS and often dies an unhappy death. Shortly after huge page support was added, it was determined that it would be better to detect a shortage of huge pages at mmap() time. The idea is that if there were not enough huge pages to cover the mapping, the mmap() would fail. This was first done with a simple check in the code at mmap() time to determine if there were enough free huge pages to cover the mapping. Like most things in the kernel, the code has evolved over time. However, the basic idea was to 'reserve' huge pages at mmap() time to ensure that huge pages would be available for page faults in that mapping. The description below attempts to describe how huge page reserve processing is done in the v4.10 kernel.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hugetlbfs_reserv.rst`, line 10); *backlink*

Unknown interpreted text role "ref".

Audience

This description is primarily targeted at kernel developers who are modifying hugetlbfs code.

The Data Structures

resv_huge_pages

This is a global (per-hstate) count of reserved huge pages. Reserved huge pages are only available to the task which reserved them. Therefore, the number of huge pages generally available is computed as (`free_huge_pages - resv_huge_pages`).

Reserve Map

A reserve map is described by the structure:

```
struct resv_map {
        struct kref refs;
        spinlock_t lock;
        struct list_head regions;
        long adds_in_progress;
        struct list_head region_cache;
        long region_cache_count;
};
```

There is one reserve map for each huge page mapping in the system. The regions list within the resv_map describes the regions within the mapping. A region is described as:

```
struct file_region {
        struct list_head link;
        long from;
        long to;
};
```

The 'from' and 'to' fields of the file region structure are huge page indices into the mapping. Depending on the type of mapping, a region in the reserv_map may indicate reservations exist for the range, or reservations do not exist.

Flags for MAP_PRIVATE Reservations

These are stored in the bottom bits of the reservation map pointer.

`#define HPAGE_RESV_OWNER    (1UL << 0)`

Indicates this task is the owner of the reservations associated with the mapping.

`#define HPAGE_RESV_UNMAPPED (1UL << 1)`

Indicates task originally mapping this range (and creating reserves) has unmapped a page from this task (the child) due to a failed COW.

Page Flags

The PagePrivate page flag is used to indicate that a huge page reservation must be restored when the huge page is freed. More details will be discussed in the "Freeing huge pages" section.

Reservation Map Location (Private or Shared)

A huge page mapping or segment is either private or shared. If private, it is typically only available to a single address space (task). If shared, it can be mapped into multiple address spaces (tasks). The location and semantics of the reservation map is significantly different for the two types of mappings. Location differences are:

- For private mappings, the reservation map hangs off the VMA structure. Specifically, vma->vm_private_data. This reserve map is created at the time the mapping (mmap(MAP_PRIVATE)) is created.
- For shared mappings, the reservation map hangs off the inode. Specifically, inode->i_mapping->private_data. Since shared mappings are always backed by files in the hugetlbfs filesystem, the hugetlbfs code ensures each inode contains a reservation map. As a result, the reservation map is allocated when the inode is created.

Creating Reservations

Reservations are created when a huge page backed shared memory segment is created (shmget(SHM_HUGETLB)) or a mapping is created via mmap(MAP_HUGETLB). These operations result in a call to the routine hugetlb_reserve_pages():

```
int hugetlb_reserve_pages(struct inode *inode,
                          long from, long to,
                          struct vm_area_struct *vma,
                          vm_flags_t vm_flags)
```

The first thing hugetlb_reserve_pages() does is check if the NORESERVE flag was specified in either the shmget() or mmap() call. If NORESERVE was specified, then this routine returns immediately as no reservations are desired.

The arguments 'from' and 'to' are huge page indices into the mapping or underlying file. For shmget(), 'from' is always 0 and 'to' corresponds to the length of the segment/mapping. For mmap(), the offset argument could be used to specify the offset into the underlying file. In such a case, the 'from' and 'to' arguments have been adjusted by this offset.

One of the big differences between PRIVATE and SHARED mappings is the way in which reservations are represented in the reservation map.

- For shared mappings, an entry in the reservation map indicates a reservation exists or did exist for the corresponding page. As reservations are consumed, the reservation map is not modified.
- For private mappings, the lack of an entry in the reservation map indicates a reservation exists for the corresponding page. As reservations are consumed, entries are added to the reservation map. Therefore, the reservation map can also be used to determine which reservations have been consumed.

For private mappings, hugetlb_reserve_pages() creates the reservation map and hangs it off the VMA structure. In addition, the HPAGE_RESV_OWNER flag is set to indicate this VMA owns the reservations.

The reservation map is consulted to determine how many huge page reservations are needed for the current mapping/segment. For private mappings, this is always the value (to - from). However, for shared mappings it is possible that some reservations may already exist within the range (to - from). See the section :ref:`Reservation Map Modifications <resv_map_modifications>` for details on how this is accomplished.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hugetlbfs_reserv.rst`, line 139); *backlink*

Unknown interpreted text role "ref".

The mapping may be associated with a subpool. If so, the subpool is consulted to ensure there is sufficient space for the mapping. It is possible that the subpool has set aside reservations that can be used for the mapping. See the section :ref:`Subpool Reservations <sub_pool_resv>` for more details.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hugetlbfs_reserv.rst`, line 146); *backlink*

Unknown interpreted text role "ref".

After consulting the reservation map and subpool, the number of needed new reservations is known. The routine hugetlb_acct_memory() is called to check for and take the requested number of reservations. hugetlb_acct_memory() calls into routines that potentially allocate and adjust surplus page counts. However, within those routines the code is simply checking to ensure there are enough free huge pages to accommodate the reservation. If there are, the global reservation count resv_huge_pages is adjusted something like the following:

```
if (resv_needed <= (resv_huge_pages - free_huge_pages))
        resv_huge_pages += resv_needed;
```

Note that the global lock hugetlb_lock is held when checking and adjusting these counters.

If there were enough free huge pages and the global count resv_huge_pages was adjusted, then the reservation map associated with the mapping is modified to reflect the reservations. In the case of a shared mapping, a file_region will exist that includes the range 'from' - 'to'. For private mappings, no modifications are made to the reservation map as lack of an entry indicates a reservation exists.

If hugetlb_reserve_pages() was successful, the global reservation count and reservation map associated with the mapping will be modified as required to ensure reservations exist for the range 'from' - 'to'.

Consuming Reservations/Allocating a Huge Page

Reservations are consumed when huge pages associated with the reservations are allocated and instantiated in the corresponding mapping. The allocation is performed within the routine alloc_huge_page():

```
struct page *alloc_huge_page(struct vm_area_struct *vma,
                             unsigned long addr, int avoid_reserve)
```

alloc_huge_page is passed a VMA pointer and a virtual address, so it can consult the reservation map to determine if a reservation exists. In addition, alloc_huge_page takes the argument avoid_reserve which indicates reserves should not be used even if it appears they have been set aside for the specified address. The avoid_reserve argument is most often used in the case of Copy on Write and Page Migration where additional copies of an existing page are being allocated.

The helper routine vma_needs_reservation() is called to determine if a reservation exists for the address within the mapping(vma). See the section :ref:`Reservation Map Helper Routines <resv_map_helpers>` for detailed information on what this routine does. The value returned from vma_needs_reservation() is generally 0 or 1. 0 if a reservation exists for the address, 1 if no reservation exists. If a reservation does not exist, and there is a subpool associated with the mapping the subpool is consulted to determine if it contains reservations. If the subpool contains reservations, one can be used for this allocation. However, in every case the avoid_reserve argument overrides the use of a reservation for the allocation. After determining whether a reservation exists and can be used for the allocation, the routine dequeue_huge_page_vma() is called. This routine takes two arguments related to reservations:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hugetlbfs_reserv.rst`, line 197); ***backlink***

Unknown interpreted text role "ref".

- avoid_reserve, this is the same value/argument passed to alloc_huge_page()
- chg, even though this argument is of type long only the values 0 or 1 are passed to dequeue_huge_page_vma. If the value is 0, it indicates a reservation exists (see the section "Memory Policy and Reservations" for possible issues). If the value is 1, it indicates a reservation does not exist and the page must be taken from the global free pool if possible.

The free lists associated with the memory policy of the VMA are searched for a free page. If a page is found, the value free_huge_pages is decremented when the page is removed from the free list. If there was a reservation associated with the page, the following adjustments are made:

```
SetPagePrivate(page);   /* Indicates allocating this page consumed
                         * a reservation, and if an error is
                         * encountered such that the page must be
                         * freed, the reservation will be restored. */
resv_huge_pages--;      /* Decrement the global reservation count */
```

Note, if no huge page can be found that satisfies the VMA's memory policy an attempt will be made to allocate one using the buddy allocator. This brings up the issue of surplus huge pages and overcommit which is beyond the scope reservations. Even if a surplus page is allocated, the same reservation based adjustments as above will be made: SetPagePrivate(page) and resv_huge_pages--.

After obtaining a new huge page, (page)->private is set to the value of the subpool associated with the page if it exists. This will be used for subpool accounting when the page is freed.

The routine vma_commit_reservation() is then called to adjust the reserve map based on the consumption of the reservation. In general, this involves ensuring the page is represented within a file_region structure of the region map. For shared mappings where the reservation was present, an entry in the reserve map already existed so no change is made. However, if there was no reservation in a shared mapping or this was a private mapping a new entry must be created.

It is possible that the reserve map could have been changed between the call to vma_needs_reservation() at the beginning of alloc_huge_page() and the call to vma_commit_reservation() after the page was allocated. This would be possible if hugetlb_reserve_pages was called for the same page in a shared mapping. In such cases, the reservation count and subpool free page count will be off by one. This rare condition can be identified by comparing the return value from vma_needs_reservation and vma_commit_reservation. If such a race is detected, the subpool and global reserve counts are adjusted to compensate. See the section :ref:`Reservation Map Helper Routines <resv_map_helpers>` for more information on these routines.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hugetlbfs_reserv.rst`, line 248);

backlink

Unknown interpreted text role "ref".

Instantiate Huge Pages

After huge page allocation, the page is typically added to the page tables of the allocating task. Before this, pages in a shared mapping are added to the page cache and pages in private mappings are added to an anonymous reverse mapping. In both cases, the PagePrivate flag is cleared. Therefore, when a huge page that has been instantiated is freed no adjustment is made to the global reservation count (resv_huge_pages).

Freeing Huge Pages

Huge page freeing is performed by the routine free_huge_page(). This routine is the destructor for hugetlbfs compound pages. As a result, it is only passed a pointer to the page struct. When a huge page is freed, reservation accounting may need to be performed. This would be the case if the page was associated with a subpool that contained reserves, or the page is being freed on an error path where a global reserve count must be restored.

The page->private field points to any subpool associated with the page. If the PagePrivate flag is set, it indicates the global reserve count should be adjusted (see the section :ref:`Consuming Reservations/Allocating a Huge Page <consume_resv>` for information on how these are set).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hugetlbfs_reserv.rst`, line 282); *backlink*

Unknown interpreted text role "ref".

The routine first calls hugepage_subpool_put_pages() for the page. If this routine returns a value of 0 (which does not equal the value passed 1) it indicates reserves are associated with the subpool, and this newly free page must be used to keep the number of subpool reserves above the minimum size. Therefore, the global resv_huge_pages counter is incremented in this case.

If the PagePrivate flag was set in the page, the global resv_huge_pages counter will always be incremented.

Subpool Reservations

There is a struct hstate associated with each huge page size. The hstate tracks all huge pages of the specified size. A subpool represents a subset of pages within a hstate that is associated with a mounted hugetlbfs filesystem.

When a hugetlbfs filesystem is mounted a min_size option can be specified which indicates the minimum number of huge pages required by the filesystem. If this option is specified, the number of huge pages corresponding to min_size are reserved for use by the filesystem. This number is tracked in the min_hpages field of a struct hugepage_subpool. At mount time, hugetlb_acct_memory(min_hpages) is called to reserve the specified number of huge pages. If they can not be reserved, the mount fails.

The routines hugepage_subpool_get/put_pages() are called when pages are obtained from or released back to a subpool. They perform all subpool accounting, and track any reservations associated with the subpool. hugepage_subpool_get/put_pages are passed the number of huge pages by which to adjust the subpool 'used page' count (down for get, up for put). Normally, they return the same value that was passed or an error if not enough pages exist in the subpool.

However, if reserves are associated with the subpool a return value less than the passed value may be returned. This return value indicates the number of additional global pool adjustments which must be made. For example, suppose a subpool contains 3 reserved huge pages and someone asks for 5. The 3 reserved pages associated with the subpool can be used to satisfy part of the request. But, 2 pages must be obtained from the global pools. To relay this information to the caller, the value 2 is returned. The caller is then responsible for attempting to obtain the additional two pages from the global pools.

COW and Reservations

Since shared mappings all point to and use the same underlying pages, the biggest reservation concern for COW is private mappings. In this case, two tasks can be pointing at the same previously allocated page. One task attempts to write to the page, so a new page must be allocated so that each task points to its own page.

When the page was originally allocated, the reservation for that page was consumed. When an attempt to allocate a new page is made as a result of COW, it is possible that no free huge pages are free and the allocation will fail.

When the private mapping was originally created, the owner of the mapping was noted by setting the HPAGE_RESV_OWNER bit in the pointer to the reservation map of the owner. Since the owner created the mapping, the owner owns all the reservations associated with the mapping. Therefore, when a write fault occurs and there is no page available, different action is taken for the owner and non-owner of the reservation.

In the case where the faulting task is not the owner, the fault will fail and the task will typically receive a SIGBUS.

If the owner is the faulting task, we want it to succeed since it owned the original reservation. To accomplish this, the page is unmapped from the non-owning task. In this way, the only reference is from the owning task. In addition, the HPAGE_RESV_UNMAPPED bit is set in the reservation map pointer of the non-owning task. The non-owning task may receive a SIGBUS if it later faults on a non-present page. But, the original owner of the mapping/reservation will behave as expected.

Reservation Map Modifications

The following low level routines are used to make modifications to a reservation map. Typically, these routines are not called directly. Rather, a reservation map helper routine is called which calls one of these low level routines. These low level routines are fairly well documented in the source code (mm/hugetlb.c). These routines are:

```
long region_chg(struct resv_map *resv, long f, long t);
long region_add(struct resv_map *resv, long f, long t);
void region_abort(struct resv_map *resv, long f, long t);
long region_count(struct resv_map *resv, long f, long t);
```

Operations on the reservation map typically involve two operations:

1. region_chg() is called to examine the reserve map and determine how many pages in the specified range [f, t) are NOT currently represented.

   The calling code performs global checks and allocations to determine if there are enough huge pages for the operation to succeed.

2. a. If the operation can succeed, region_add() is called to actually modify the reservation map for the same range [f, t) previously passed to region_chg().
   b. If the operation can not succeed, region_abort is called for the same range [f, t) to abort the operation.

Note that this is a two step process where region_add() and region_abort() are guaranteed to succeed after a prior call to region_chg() for the same range. region_chg() is responsible for pre-allocating any data structures necessary to ensure the subsequent operations (specifically region_add())) will succeed.

As mentioned above, region_chg() determines the number of pages in the range which are NOT currently represented in the map. This number is returned to the caller. region_add() returns the number of pages in the range added to the map. In most cases, the return value of region_add() is the same as the return value of region_chg(). However, in the case of shared mappings it is possible for changes to the reservation map to be made between the calls to region_chg() and region_add(). In this case, the return value of region_add() will not match the return value of region_chg(). It is likely that in such cases global counts and subpool accounting will be incorrect and in need of adjustment. It is the responsibility of the caller to check for this condition and make the appropriate adjustments.

The routine region_del() is called to remove regions from a reservation map. It is typically called in the following situations:

- When a file in the hugetlbfs filesystem is being removed, the inode will be released and the reservation map freed. Before freeing the reservation map, all the individual file_region structures must be freed. In this case region_del is passed the range [0, LONG_MAX).
- When a hugetlbfs file is being truncated. In this case, all allocated pages after the new file size must be freed. In addition, any file_region entries in the reservation map past the new end of file must be deleted. In this case, region_del is passed the range [new_end_of_file, LONG_MAX).
- When a hole is being punched in a hugetlbfs file. In this case, huge pages are removed from the middle of the file one at a time. As the pages are removed, region_del() is called to remove the corresponding entry from the reservation map. In this case, region_del is passed the range [page_idx, page_idx + 1).

In every case, region_del() will return the number of pages removed from the reservation map. In VERY rare cases, region_del() can fail. This can only happen in the hole punch case where it has to split an existing file_region entry and can not allocate a new structure. In this error case, region_del() will return -ENOMEM. The problem here is that the reservation map will indicate that there is a reservation for the page. However, the subpool and global reservation counts will not reflect the reservation. To handle this situation, the routine hugetlb_fix_reserve_counts() is called to adjust the counters so that they correspond with the reservation map entry that could not be deleted.

region_count() is called when unmapping a private huge page mapping. In private mappings, the lack of a entry in the reservation map indicates that a reservation exists. Therefore, by counting the number of entries in the reservation map we know how many reservations were consumed and how many are outstanding (outstanding = (end - start) - region_count(resv, start, end)). Since the mapping is going away, the subpool and global reservation counts are decremented by the number of outstanding reservations.

Reservation Map Helper Routines

Several helper routines exist to query and modify the reservation maps. These routines are only interested with reservations for a specific huge page, so they just pass in an address instead of a range. In addition, they pass in the associated VMA. From the VMA, the type of mapping (private or shared) and the location of the reservation map (inode or VMA) can be determined. These routines simply call the underlying routines described in the section "Reservation Map Modifications". However, they do take into account the 'opposite' meaning of reservation map entries for private and shared mappings and hide this detail from the caller:

```
long vma_needs_reservation(struct hstate *h,
                           struct vm_area_struct *vma,
                           unsigned long addr)
```

This routine calls region_chg() for the specified page. If no reservation exists, 1 is returned. If a reservation exists, 0 is returned:

```
long vma_commit_reservation(struct hstate *h,
                            struct vm_area_struct *vma,
                            unsigned long addr)
```

This calls region_add() for the specified page. As in the case of region_chg and region_add, this routine is to be called after a previous call to vma_needs_reservation. It will add a reservation entry for the page. It returns 1 if the reservation was added and 0 if not. The return value should be compared with the return value of the previous call to vma_needs_reservation. An unexpected difference indicates the reservation map was modified between calls:

```
void vma_end_reservation(struct hstate *h,
                         struct vm_area_struct *vma,
                         unsigned long addr)
```

This calls region_abort() for the specified page. As in the case of region_chg and region_abort, this routine is to be called after a previous call to vma_needs_reservation. It will abort/end the in progress reservation add operation:

```
long vma_add_reservation(struct hstate *h,
                         struct vm_area_struct *vma,
                         unsigned long addr)
```

This is a special wrapper routine to help facilitate reservation cleanup on error paths. It is only called from the routine restore_reserve_on_error(). This routine is used in conjunction with vma_needs_reservation in an attempt to add a reservation to the reservation map. It takes into account the different reservation map semantics for private and shared mappings. Hence, region_add is called for shared mappings (as an entry present in the map indicates a reservation), and region_del is called for private mappings (as the absence of an entry in the map indicates a reservation). See the section "Reservation cleanup in error paths" for more information on what needs to be done on error paths.

Reservation Cleanup in Error Paths

As mentioned in the section :ref:`Reservation Map Helper Routines <resv_map_helpers>`, reservation map modifications are performed in two steps. First vma_needs_reservation is called before a page is allocated. If the allocation is successful, then vma_commit_reservation is called. If not, vma_end_reservation is called. Global and subpool reservation counts are adjusted based on success or failure of the operation and all is well.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\vm\[linux-master][Documentation][vm]hugetlbfs_reserv.rst`, line 514); *backlink*

Unknown interpreted text role "ref".

Additionally, after a huge page is instantiated the PagePrivate flag is cleared so that accounting when the page is ultimately freed is correct.

However, there are several instances where errors are encountered after a huge page is allocated but before it is instantiated. In this case, the page allocation has consumed the reservation and made the appropriate subpool, reservation map and global count adjustments. If the page is freed at this time (before instantiation and clearing of PagePrivate), then free_huge_page will increment the global reservation count. However, the reservation map indicates the reservation was consumed. This resulting inconsistent state will cause the 'leak' of a reserved huge page. The global reserve count will be higher than it should and prevent allocation of a pre-allocated page.

The routine restore_reserve_on_error() attempts to handle this situation. It is fairly well documented. The intention of this routine is to restore the reservation map to the way it was before the page allocation. In this way, the state of the reservation map will correspond to the global reservation count after the page is freed.

The routine restore_reserve_on_error itself may encounter errors while attempting to restore the reservation map entry. In this case, it will simply clear the PagePrivate flag of the page. In this way, the global reserve count will not be incremented when the page is freed. However, the reservation map will continue to look as though the reservation was consumed. A page can still be allocated for the address, but it will not use a reserved page as originally intended.

There is some code (most notably userfaultfd) which can not call restore_reserve_on_error. In this case, it simply modifies the PagePrivate so that a reservation will not be leaked when the huge page is freed.

Reservations and Memory Policy

Per-node huge page lists existed in struct hstate when git was first used to manage Linux code. The concept of reservations was added some time later. When reservations were added, no attempt was made to take memory policy into account. While cpusets are not exactly the same as memory policy, this comment in hugetlb_acct_memory sums up the interaction between reservations and cpusets/memory policy:

```
/*
 * When cpuset is configured, it breaks the strict hugetlb page
 * reservation as the accounting is done on a global variable. Such
 * reservation is completely rubbish in the presence of cpuset because
 * the reservation is not checked against page availability for the
 * current cpuset. Application can still potentially OOM'ed by kernel
 * with lack of free htlb page in cpuset that the task is in.
 * Attempt to enforce strict accounting with cpuset is almost
 * impossible (or too ugly) because cpuset is too fluid that
 * task or memory node can be dynamically moved between cpusets.
 *
 * The change of semantics for shared hugetlb mapping with cpuset is
 * undesirable. However, in order to preserve some of the semantics,
 * we fall back to check against current free page availability as
 * a best attempt and hopefully to minimize the impact of changing
 * semantics that cpuset has.
 */
```

Huge page reservations were added to prevent unexpected page allocation failures (OOM) at page fault time. However, if an application makes use of cpusets or memory policy there is no guarantee that huge pages will be available on the required nodes. This is true even if there are a sufficient number of global reservations.

Hugetlbfs regression testing

The most complete set of hugetlb tests are in the libhugetlbfs repository. If you modify any hugetlb related code, use the libhugetlbfs test suite to check for regressions. In addition, if you add any new hugetlb functionality, please add appropriate tests to libhugetlbfs.

-- Mike Kravetz, 7 April 2017