guest_memfd HugeTLB support overview

For 2025-10-30 guest_memfd bi-weekly upstream call

Contact ackerleytng@google.com if you have questions/suggestions!

Discussion overview

- What is guest_memfd with HugeTLB?
- Phased introduction in ~3 patch series + 1 prerequisite fix series
  - Required modifications to HugeTLB
    - How guest_memfd will use the modifications to HugeTLB
  - uAPI discussion
  - Code organization

What is guest_memfd with HugeTLB?

- Support for up to 1G page table mappings in stage 2 page tables
- Get huge pages from HugeTLB because
  - HugeTLB will be the central place to manage memory on host machines for co-tenancy of
    - non-CoCo VMs already using HugeTLB with
    - CoCo VMs that have to use guest_memfd and need huge pages
  - Benefit from HugeTLB vmemmap optimization
- Will respect all of HugeTLB’s quotas, reservations, (cgroup) charging, etc

Proposal: Phased introduction in [stages]

- [st_blocks] Track allocations in guest_memfd, have guest_memfd update `st_blocks` in `fstat()`
  - ~4 patches (1 of those selftests)
- [HugeTLB support] Add HugeTLB support for private-only guest_memfd
  - No INIT_SHARED, no restructuring
  - ~30 patches (~15 of those selftests)
- [HugeTLB restructuring] Let HugeTLB be used with mmap() and conversions
  - Restructuring: split directly from `PUD_SIZE` to `PAGE_SIZE` and merge directly back
  - ~25 patches (8 of those selftests)
- [Optimizing restructuring] Restructuring includes `PMD_SIZE`
  - ~10 patches (# selftests TBD)

[st_blocks]	[HugeTLB support]	[HugeTLB restructuring]	[Optimizing restructuring]

[st_blocks]

What/why `st_blocks`?

- Currently for (`PAGE_SIZE` guest_memfd), `st_blocks` is always 0
  - Allocations don't change `st_blocks` in `fstat()`
- Why update this?
  - tmpfs and HugeTLBfs both update `st_blocks`
    - guest_memfd should be consistent
  - `st_blocks` lets userspace track actual memory usage per-file
    - Cgroup usage tracking could work, but that's at the process/cgroup level
- Why is this a prerequisite?
  - Not directly, but the implementation lends itself well to being used for HugeTLB

Implementation

- After allocating a folio for guest_memfd, increment `inode->i_blocks`
- => Need custom truncation function to decrement `inode->i_blocks`
- (Custom truncation function needed for HugeTLB+restructuring support)

[HugeTLB support]

Recap: usage modes after Sean's mmap fixes series

guest_memfd only for private memory

- KVM module param `vm_memory_attributes` is true
- Shared/private status tracked at VM level
- Conversions via `SET_MEMORY_ATTRIBUTES` VM ioctl
- aka legacy dual backing

guest_memfd for shared+private

- `vm_memory_attributes = false`
- Shared/private status tracked in guest_memfd
- guest_memfd ioctl
- aka single backing

HugeTLB feature availability

guest_memfd only for private memory

- Can use for private memory, shared memory must be from somewhere else
- `INIT_SHARED` and `HUGETLB` will be mutually exclusive (`EINVAL`)
  - To disable host faults, since memory will be all private
- guest_memfd conversion ioctls already fail since `vm_memory_attributes` allow the VM ioctl and disable the guest_memfd ioctl

guest_memfd for shared+private

- `HUGETLB` flag won't be among the valid flags returned from `KVM_CAP_GUEST_MEMFD_FLAGS`

Why no `INIT_SHARED` with `HUGETLB`?

- Must disable faulting of HugeTLB pages
- guest_memfd uses core-mm's fault handler, which does not handle mapping of HugeTLB pages
    - Unsetting the HugeTLB folio flag will interfere with HVO in guest_memfd
        - HVO functions check for the HugeTLB folio flag
- I'm assuming having core-mm map 1G pages is going to be complex, does anyone know otherwise?
    - Giving core-mm a HugeTLB folio results in WARN()
    - Have not explored just unsetting the HugeTLB folio flag

What's the point of this stage?

- To introduce the feature in stages, reduce per-stage complexity
- Still useful for legacy dual-backing setups
- Important for introducing/exercising/testing:
  - KVM's mapping at higher page levels than `PAGE_SIZE`
    - KVM's being limited to map at lower page levels even for huge pages, when `base_gfn` is not aligned with guest_memfd offset.
  - HugeTLB quota/reservations/statistics, required refactoring

HugeTLB folio usage in guest_memfd (1)

- guest_memfd creation time
  - Validate order: only valid HugeTLB orders allowed
  - Create subpool just for this guest_memfd
  - Subpool size == guest_memfd size
  - guest_memfd will never use surplus HugeTLB pages
  - Charge reservations to cgroup on subpool creation
    - => Reservations charged on guest_memfd creation time
- Folio allocation time
  - Charge usage to cgroup
  - => Charge usage at allocation time, may be different cgroup from reservations
    - Same as HugeTLB: reservations charged at mmap() time and usage charged at allocation time

HugeTLB folio usage in guest_memfd (2)

- `fallocate()`: Allocate and punch hole at granularities smaller than HugeTLB page size => `EINVAL`
- KVM faults (`kvm_gmem_get_pfn()`)
  - Return huge page, with `max_order = folio_order(folio)`
- Host faults disabled since memory always private (`INIT_SHARED` is disabled)
- Memory failure: *should* be supported via regular HugeTLB memory failure handling
  - TODO: testing this

HugeTLB refactoring: `alloc_hugetlb_folio()`

HugeTLBfs

- Subpool stored on filesystem mount
- Reservations stored on VMA
- Memory policy passed via `vma->vm_policy`
- VMA required for allocation
  - HugeTLBfs uses a pseudo-vma

guest_memfd

- Subpool is per-inode, stored on inode
- No VMA reservations
- Memory policy stored on inode, awkward to pass via (pseudo) VMA
- Pages may not be mapped to userspace => no VMAs

HugeTLB refactoring: `alloc_hugetlb_folio()`

- Refactor to accept
  - hstate
  - memory policy
  - interleaving index
  - whether to charge reservations to cgroup
  - whether to use existing reservations in hstate

HugeTLB refactoring: `alloc_hugetlb_folio()`

- Refactor to accept
  - hstate
  - memory policy
  - interleaving index
  - whether to charge reservations to cgroup
  - whether to use existing reservations in hstate

Code organization

- All functions will be under `virt/kvm/`
- All functions built as part of the KVM module
- Export HugeTLB functions for KVM module
- Some HugeTLB function refactoring to encapsulate management of `hstates` as an array
  - Concept of `struct hstate *` will still be used

Advertising support

- `vm_memory_attributes` are supported
  - `KVM_CAP_GUEST_MEMFD_FLAGS` will include `GUEST_MEMFD_FLAG_HUGETLB`
- `vm_memory_attributes = false`
  - `KVM_CAP_GUEST_MEMFD_FLAGS` will NOT include `GUEST_MEMFD_FLAG_HUGETLB`

- `INIT_SHARED` and `HUGETLB` are mutually exclusive

Known Issues

- Race during allocation of the same hugepage offset
  - Losing thread will get `ENOMEM` since subpool is exhausted
  - HugeTLB: losing thread will get a surplus page, realize it lost the race, free the surplus page and share the page with the winning thread
    - (guest_memfd doesn't support surplus pages)
- Untested memory failure handling

[HugeTLB restructuring]

What does restructuring give us?

- Being able to fault (split) HugeTLB pages
- => Can convert from private to shared
- `INIT_SHARED` and `HUGETLB` remain mutually exclusive

Why still no INIT_SHARED with HUGETLB?

- Use case pointless?
- INIT_SHARED with HUGETLB means every fault will result in split pages
- If pages are always used as split => just use PAGE_SIZE guest_memfd
- Still possible to start all private and convert all to shared, but inefficient
- Future: restriction would probably be unlocked when core-mm can fault at higher levels from guest_memfd
    - Would be useful for non-CoCo VMs

When do we restructure?

- Split if any page ranges are shared at allocation time
- On conversion from private to shared, split pages to `PAGE_SIZE`
- On conversion from shared to private, merge pages to original size if entire range is private
- Merge on truncation of entire page
    - i.e. merge on removal of huge page from guest_memfd ownership

How do we restructure?

- Use `__split_folio()` from kernel
  - Add on undoing and re-application of HugeTLB Vmemmap Optimization (HVO)
- Currently experimenting with keeping folio in filemap during restructuring, using `xa_split_order()`
  - Signal boosting my question on xarrays here, would like help [1]

[1] https://lore.kernel.org/all/20251028223414.299268-1-ackerleytng@google.com/

Conversion flow (TBD)

- On a per-HugeTLB page basis
  - Lock out all (other) allocations, faults, truncations, conversions
  - For shared to private conversions
    - Unmap range from host userspace
    - Fail with `EAGAIN` if there are elevated refcounts
  - Allocate memory for updating shared/private state in maple tree => may fail with `ENOMEM`
  - Restructure folios - splitting needs allocations => may fail with `ENOMEM`
  - — point of no return —
  - Set KVM’s invalidation range up to the huge page (expanded from conversion range)
  - Split boundary leaves (TDX)
  - Unmap from stage 2 page tables
  - Commit updates to shared/private state in maple tree
  - Clean up invalidation range

Truncation quirks (hence custom truncation from [st_blocks])

- Truncation requires merging, merging requires safe refcounts
  - Independent of conversion: conversion requires safe refcounts because we don't want the host (holding the elevated refcount) to have access to private memory
  - Why not just sum refcounts?
  - Merging requires safe refcounts because the holder of the refcount is expecting a split page, might cause problems if the page were suddenly larger than expected
- `fallocate(PUNCH_HOLE)` will fail (`EAGAIN`) if there are elevated refcounts
  - Because we can fail `fallocate()`, and it is easier to handle
- But we can't fail truncations due to inode release, hence hook folio_put() to do the merging
  - Merging is complex, hence defer to kernel worker thread

Truncation on inode release

- Inode release: always try to merge in process context
- If some split page is still pinned, let folio_put() merge it
  - => folio outlives inode
- folio_put() will be called on each split page (eventually)
- Track pages yet to be merged in global data structure
  - When pages yet to be merged == original folio nr_pages, do the merge
- Do the merge in a kernel worker thread (deferred)
  - Because `folio_put()` can be called from atomic context
  - TODO: Yan’s suggestion to merge if in process context is great, does anyone know how best to check if code is called atomic/process context? [1]

[1] https://lore.kernel.org/all/diqzcy7d60e2.fsf@google.com/

Why track allocated HugeTLB folio metadata?

- Know the original size of the folio, when folio outlives inode
- Fewer fields to save/restore during folio restructuring
  - `__split_folio()` doesn't care about HugeTLB fields on the third `struct page`
  - Track metadata at allocation, restore on free
    - While folio is owned by guest_memfd these fields are static anyway
- Memory failure will use this to answer the question "Does this pfn belong to guest_memfd HugeTLB?" => handle guest_memfd HugeTLB failure separately from other folio types
- When offlining HugeTLB cgroups, `hugetlb_cgroup_css_offline()` will iterate `h->hugepage_activelist` to move HugeTLB charges (nr_pages) to parent
  - Split folio => `folio_nr_pages()` is less than it should be
  - => charge for full number of pages will not be moved
  - => look up this tracking to move charges, update charged cgroup

How to track allocated HugeTLB folio metadata?

- Multi-index XArray nicely lends itself to tracking folios by order
- Has to be tracked in the kernel itself, outside of KVM, since folio can outlive inode (and KVM)

Code organization

- Have tracking and restructuring code in `mm/hugetlb_restructuring.c`
    - (Name suggestions?)
- Built into the kernel, not in KVM module

Advertising support

- `vm_memory_attributes` are supported
  - `KVM_CAP_GUEST_MEMFD_FLAGS` will include `GUEST_MEMFD_FLAG_HUGETLB`
- `vm_memory_attributes = false`
  - `KVM_CAP_GUEST_MEMFD_FLAGS` will now include `GUEST_MEMFD_FLAG_HUGETLB`

- `INIT_SHARED` and `HUGETLB` remain mutually exclusive

Issues/Questions

- Should mmap() return a `PUD_SIZE` aligned address for guest_memfd `PUD_SIZE`-d HugeTLB?

[Optimizing restructuring]

Why optimize restructuring?

- Splitting to `PAGE_SIZE` upon sharing loses too much HVO
  - Based on usage patterns of shared pages, optimizing could save ~160MB per VM
    - => up to 40 VMs => 6GB savings
  - Some more savings from fewer DPAMT entries (TDX)
- Splitting to `PMD_SIZE` is faster than splitting to `PAGE_SIZE`
  - Numbers TBD

How to optimize restructuring?

- Upon conversion to shared, keep as many pages at `PMD_SIZE`
  - Split only the ones converted to shared to `PAGE_SIZE`
- Same for merge, merge to as large a page size as possible, based on shared/private status

Complexities

- HVO code today always removes optimization by splitting from `PUD_SIZE` to `PAGE_SIZE` and optimizes back directly
- Can achieve what we want (but inefficient)
  - `PUD_SIZE` => `PAGE_SIZE` => `PMD_SIZE`
- Vishal has done some investigation on direct `PUD_SIZE` to `PMD_SIZE` HVO re-optimization