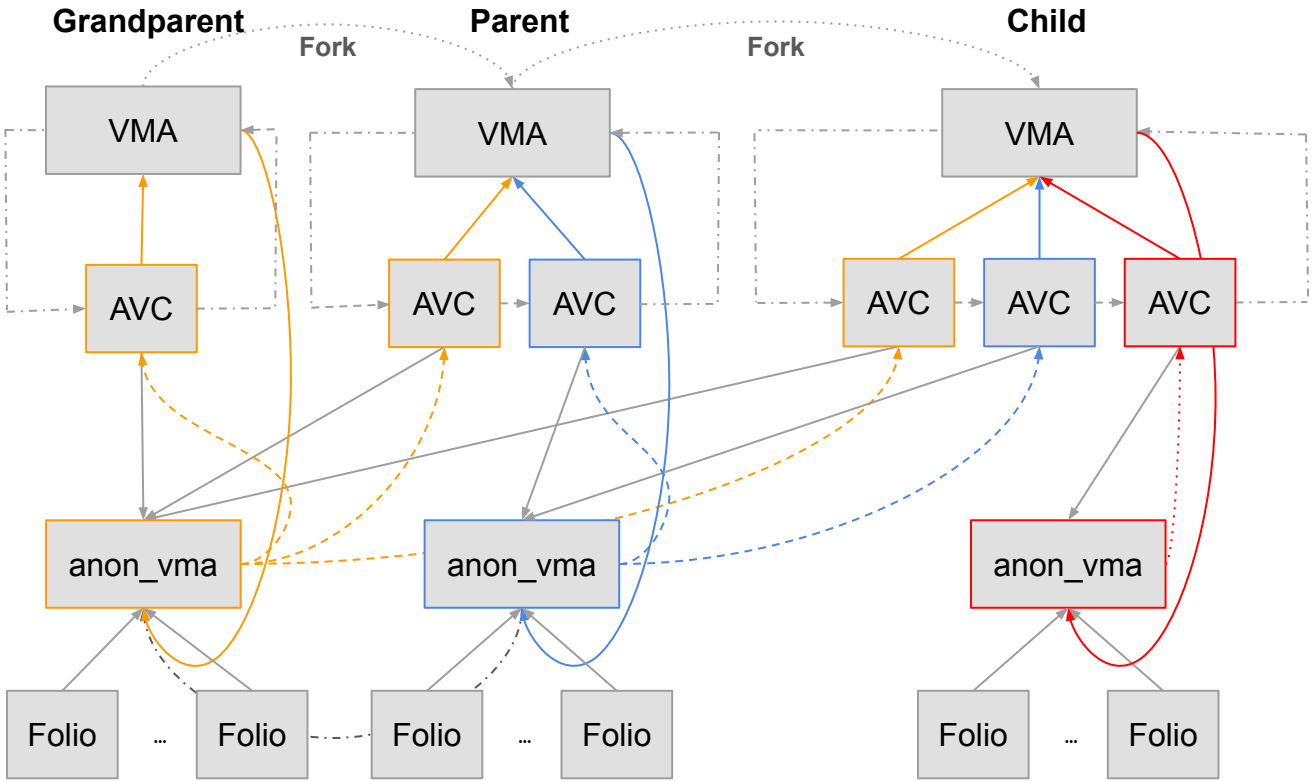
Scalable [anon] CoW

Lorenzo Stoakes (Oracle)

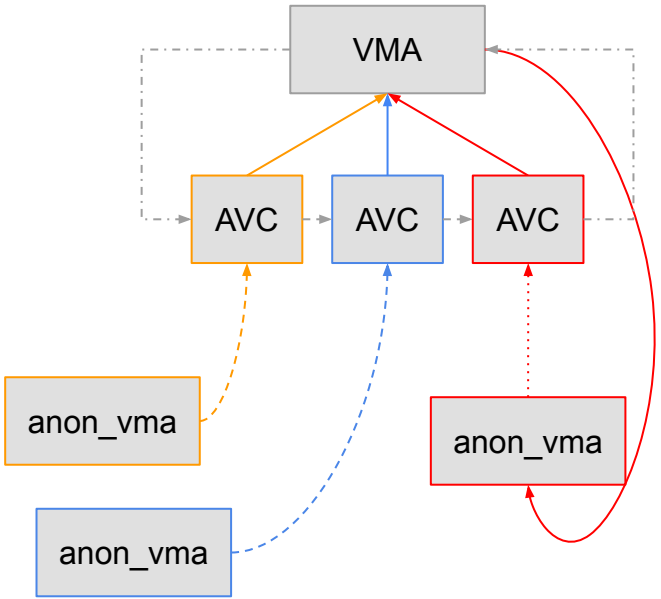
LSF/MM 2026

Forkin' hierarchy



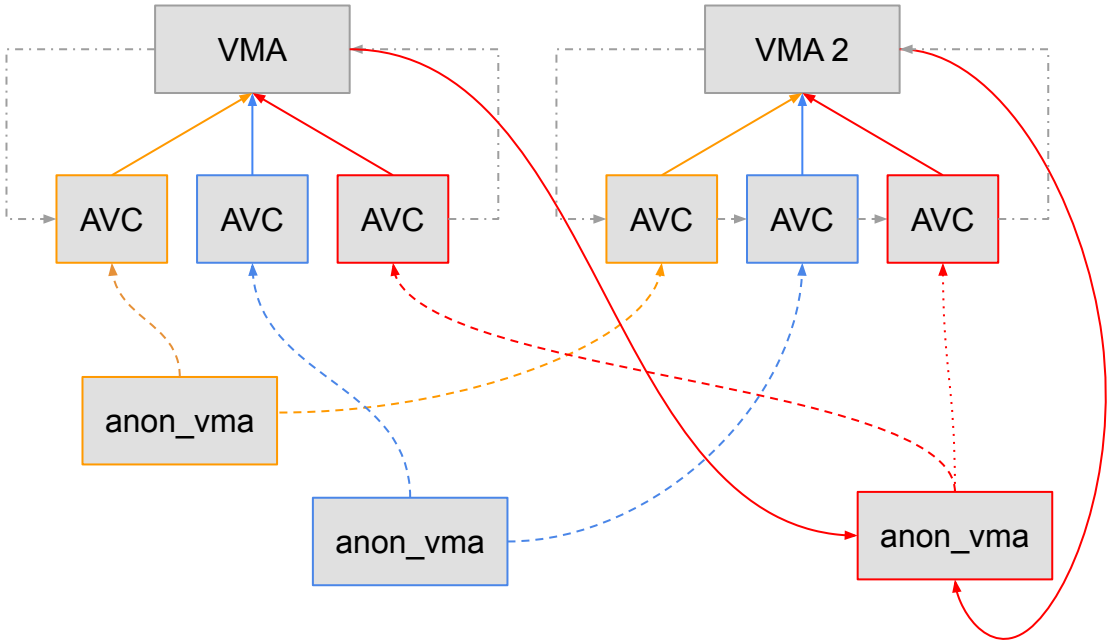
VMA manipulation

E.g. split:

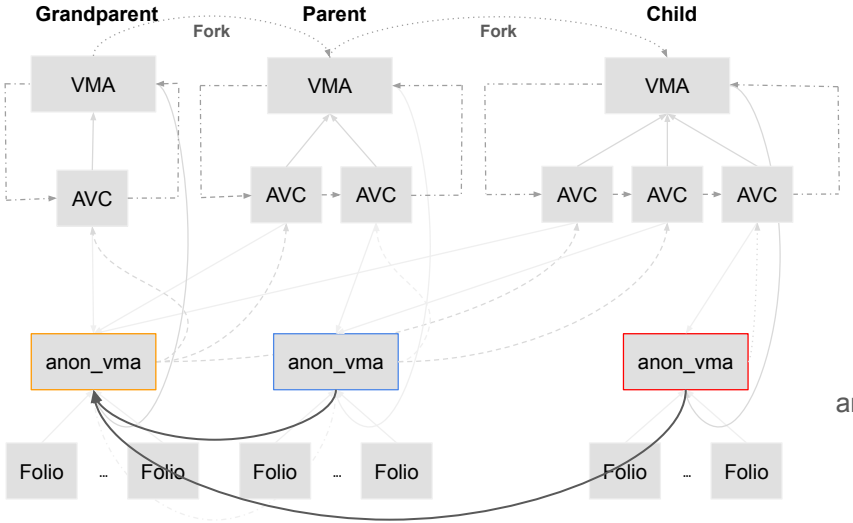


Update by:

- Deleting interval tree edges.
- Cloning AVC's to new VMA (with an extra anon_vma and AVC to connect it if forking).
- Reestablishing interval tree edges to reflect new VMA ranges.



anon_vma locking



- No matter where you are in the hierarchy, the **anon_vma lock** is **taken on** the **entire hierarchy**.
- this is done by taking a **lock** on the **root anon_vma** and tracking that in each anon_vma.
- **Fork, process tear down and VMA modification write-locks** the **entire hierarchy!**
- Which causes **rmap walks to contend**.

We **prevent unnecessary *walks*** by moving a folio to a new anon_vma on anon exclusive, but we **do not prevent unnecessary *locking***.

anon_vma **write** lock taken on:

- **Fork** (all VMAs)
- **Process teardown** (all VMAs)
- **VMA manipulation** (inherited VMAs) - mprotect(), madvise(), etc.)

anon_vma **read** lock taken on:

- **Migration** - Latency if contention.
- **THP collapse** - Latency if contention.
- **Reclaim** - this is particularly problematic, as lock contention causes folio rotation.

Problems with anon_vma

- **Memory usage**
  - anon_vma 96 bytes, anon_vma_chain 64 bytes.
  - Even lightly loaded systems have 10's of thousands of these objects.
  - Scales up under unfortunate workloads.
- **Lock contention**
  - Locking the entire rmap hierarchy is highly problematic and causes contention.
  - Large mappings can cause you problems anyway.
  - Easy to get lock inversion, limits what we can do on rmap walk.
- **Inscrutable code**
  - Code is an entirely broken abstraction that is very hard to reason about.
  - Lots of gotchas, also very fiddly reuse logic.
  - anon_vma objects have to track a **lot** of state.

Proposed alternative: CoW Context

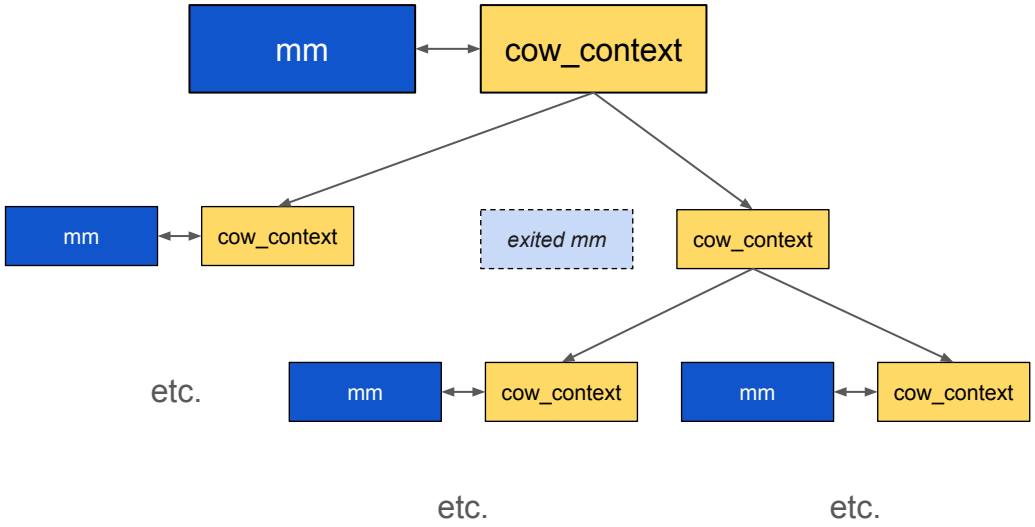
anon_vma tracks at **mapping granularity**.

Can we abstract at the mm level?

We must be able to:

- Find **fork children.**
- Look up mapped **virtual address ranges**.
- **Track fork children** after **mm teardown**.

Call this abstraction the **CoW context**.



Locking

anon_vma

Long-lived Write locks on:

- **Fork** (all VMAs)
- **Process teardown** (all VMAs)
- **VMA manipulation** (inherited VMAs) - mprotect(), madvise(), etc.)

Long-lived Read locks on:

- **Migration** - Latency if contention.
- **THP collapse** - Latency if contention.
- **Reclaim** - this is particularly problematic, as lock contention causes folio rotation. Also impacted by latency on unmap.

Across entire **fork hierarchy**.

Cow Context

Short-lived write spinlocks on:

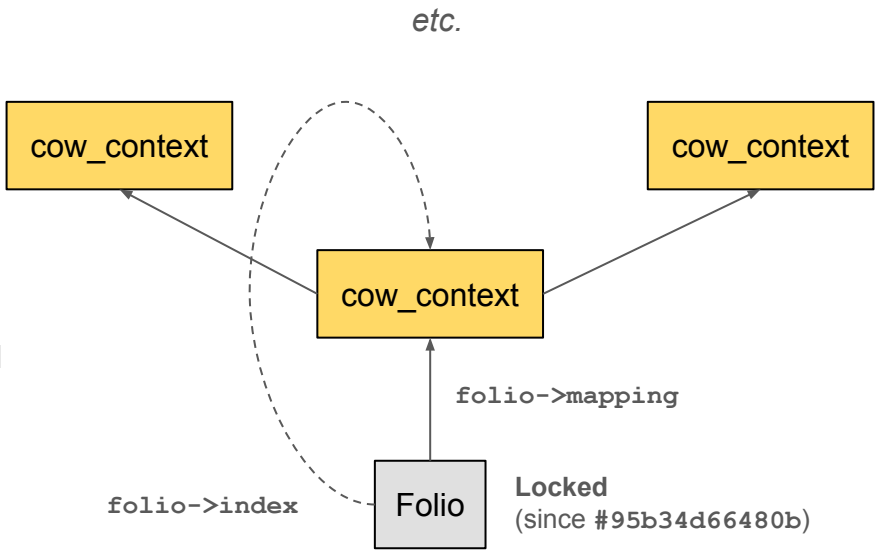
- VMA **remapping**.
- VMA mapping, unmapping and **shrinking/expansion** only **if** the VMA was **ever remapped**.
- Everything else is under **RCU** - **readers take no locks**.

However - huge **TODO** on stabilisation, see next slide.

Locks are held across a **single mapping** (remap entry).

Reverse map walk

1. (Somehow :) **Stabilise** things as needed.
2. (Somehow :) **Lookup** 'related' VMAs from **folio->index**.
3. If present in **current mm**, **walk page tables** as normal and find 'related' mappings.
4. Otherwise, **walk fork children** and examine VMAs there.
5. **Recurse** (not really it's the kernel, so iterative equivalent)



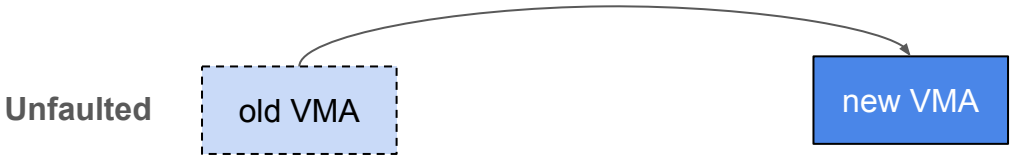
Remaps - private, anon folios

Buuuut... **remaps**.

```
mremap(from, size, new_size, MREMAP_MAYMOVE | MREMAP_FIXED, to);
```

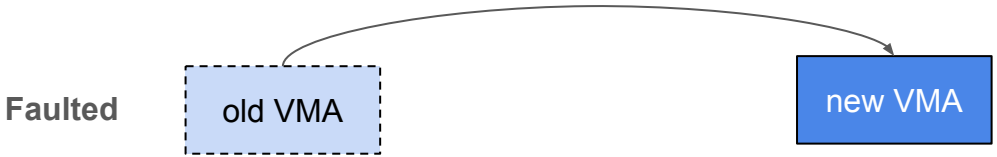
Update page offset

`new_vma->vm_pgoff = new_addr >> PAGE_SHIFT`



Keep page offset the same

`new_vma->vm_pgoff = old_vma->vm_pgoff`

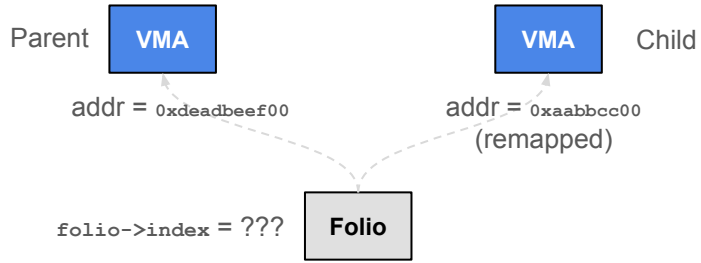


When **unfaulted**, no folios can possibly look us up from the rmap, so we **can update** `vma->vm_pgoff`.

Once **faulted**, folios need to look up mappings from the rmap, so we **cannot update** `vma->vm_pgoff`.

Expensive & tricky to update folio->index on remap.

...annnd what about forking?



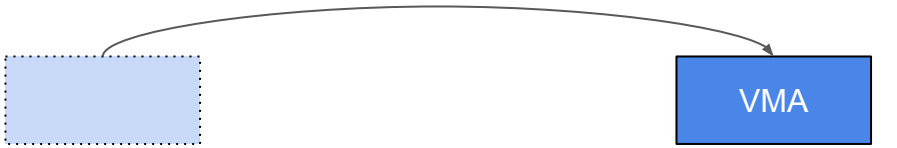
We need `vma->vm_pgoff` to disambiguate either way.

Remaps - page offset can be duplicated after remap

```
mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
```

(fault in)

```
mremap(addr, size, size, MREMAP_MAYMOVE | MREMAP_FIXED, addr + 0x100000);
```



`vma->vm_pgoff = addr>>PAGE_SHIFT`

```
mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
```

(fault in)

`vma2->vm_pgoff = addr>>PAGE_SHIFT`

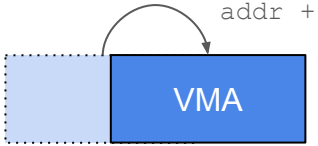
`vma->vm_pgoff = addr>>PAGE_SHIFT`

Remaps - Page offset can overlap after remap

```
mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
```

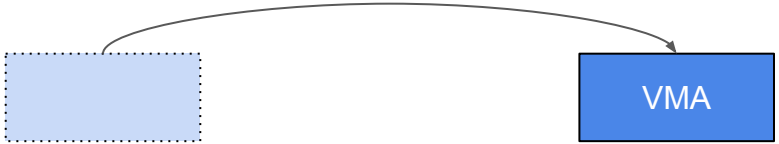
(*don't* fault in)

```
mremap(addr, size, size,
       MREMAP_MAYMOVE | MREMAP_FIXED,
           addr + size / 2);
```



(fault in)

```
mremap(addr + size / 2, size, size,
       MREMAP_MAYMOVE | MREMAP_FIXED,addr + 0x100000);
```




```
vma->vm_pgoff =
(addr + size / 2) >> PAGE_SHIFT
```

```
mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_ANON | MAP_PRIVATE, -1, 0);
```

(fault in)

```
vma2->vm_pgoff = addr >> PAGE_SHIFT
```


```
vma->vm_pgoff = (addr + size / 2) >> PAGE_SHIFT
```

Remap tracking

Key idea: **track remaps**.

Use data structure to map folio->index **ranges** to VMA **ranges**.

What data structure do we have that's good for tracking ranges like that? 🤔

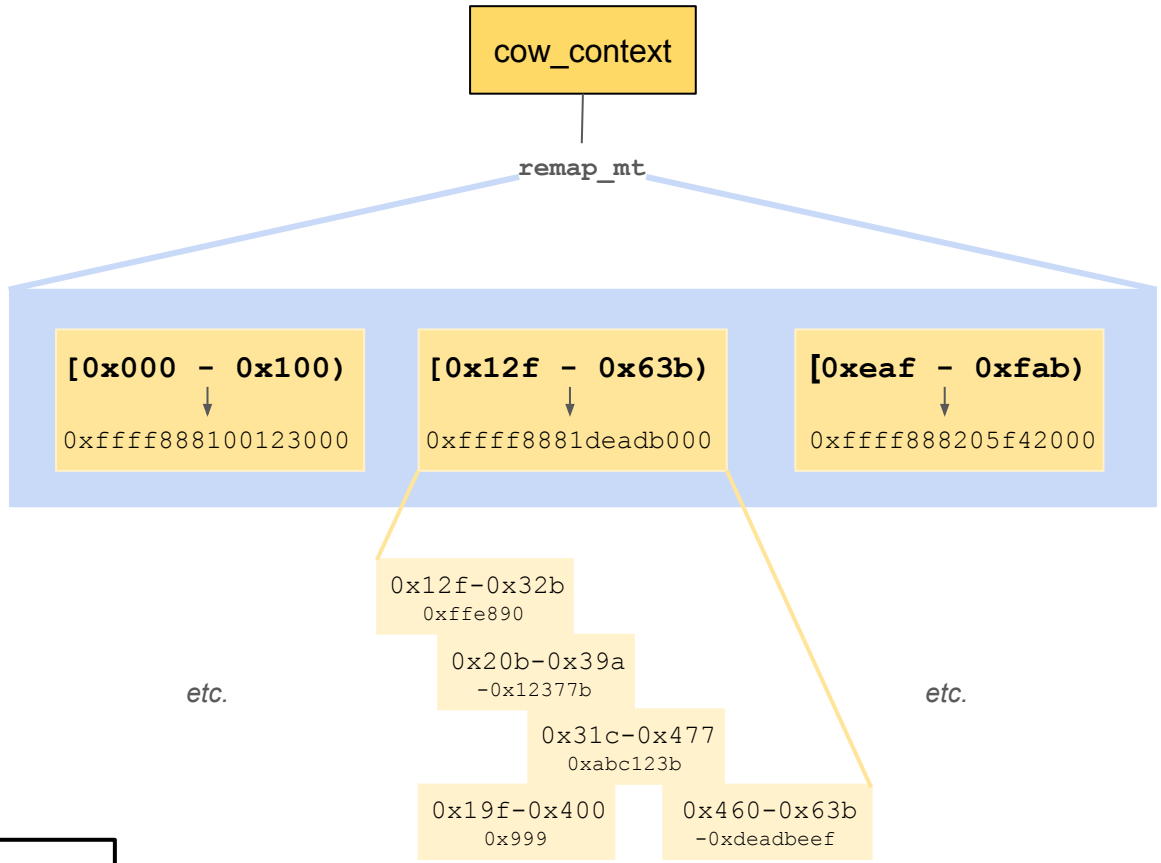
Ah yeah a **maple tree**!

But it maps **ranges** to **pointers** and doesn't account for **overlapped** (or **duplicate**) entries.

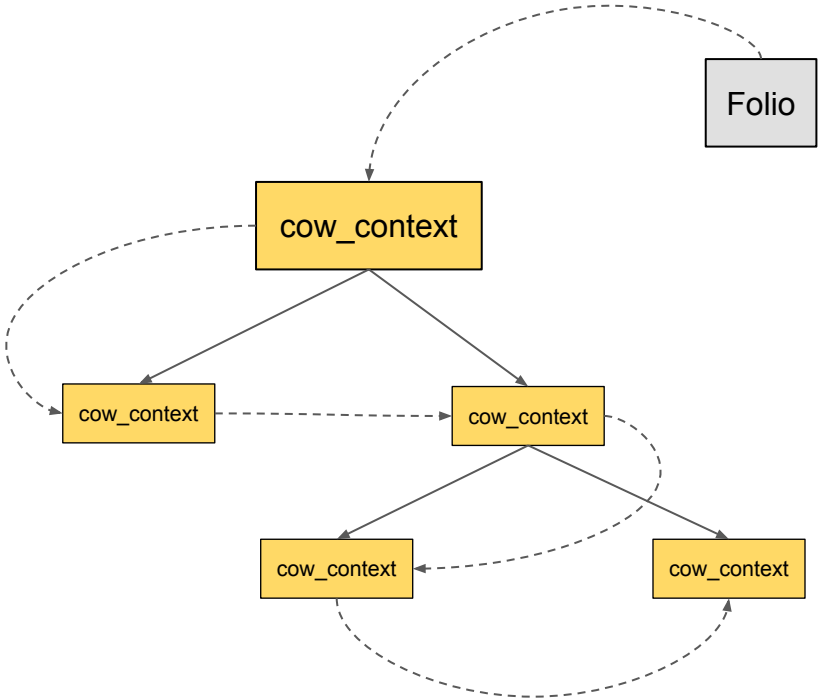
So **supplement** to allow - PoC uses a simple dynamic array under the maple tree.

Map **page offset** ranges to **offset between** `vma->vm_pgoff` and `vma->vm_start >> PAGE_SHIFT`.

∴ `addr = (folio->index + offset) << PAGE_SHIFT`



A lot of walking. Problem?



- Not tracking *actual* fork hierarchy means shared folios have to walk every possible child (could be a lot).

However:

- Like anon_vma, on becoming **anon exclusive**, we **move** the folio to the lowest cow_context level.
- **Majority** of anon memory on a system is **anon exclusive**.
- **Majority** of **shared** anon memory will be shared across ***all*** **children**.
- **Because** - you likely **fork early**, if you **CoW memory** you likely do it **after** any **fork** and don't fork after that.
- **Work** is then usually done in the fork **leaves**.
- the rest are edge cases, and being slower is ok for them.

MAP_PRIVATE files

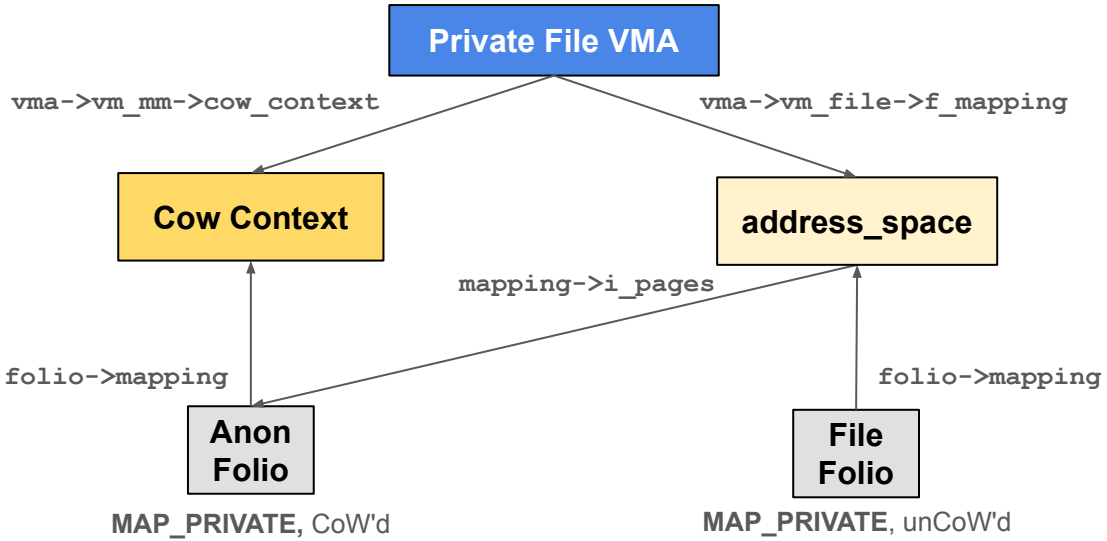
```
static inline pgoff_t
linear_page_index(const struct vm_area_struct *vma,
                  const unsigned long address)
{
        pgoff_t pgoff;
        pgoff = (address - vma->vm_start) >> PAGE_SHIFT;
        pgoff += vma->vm_pgoff;
        return pgoff;
}
```

vma->vm_pgoff = page offset of start of file

∴ folio->index = page offset into file

```
fd = open("foo.txt", O_RDWR);
ptr = mmap(NULL, size, PROT_READ | PROT_WRITE,
           MAP_PRIVATE, fd,
           pg_offset * PAGE_SIZE);

ptr[x] = 'y'; // CoW page
```

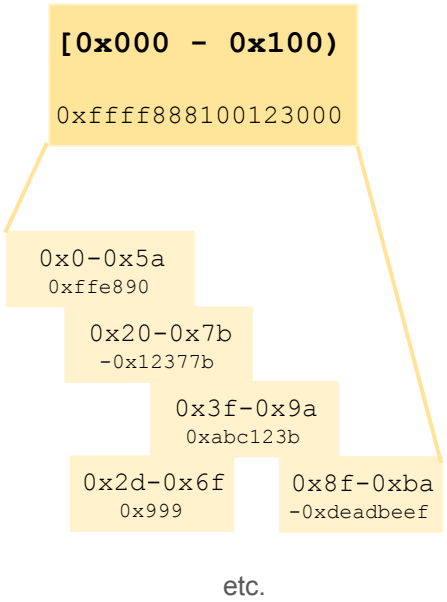


Awkward case of a **file-backed** VMA and an **anonymous** folio.

```
!vma_is_anonymous(vma)
        &&
folio_test_anon(folio)
```

Problem: MAP_PRIVATE file remap tracking

- PoC simply handles them by doing remap tracking with a *lot* of overlaps.
- Potential O(n) scaling (n=# mapped files), needs addressing.
- **Solution:** Feasible to simply set `folio->index = addr >> PAGE_SHIFT` and add a new `vma->vm_anon_pgoff` field which is set to `vma->vm_start >> PAGE_SHIFT` on first fault + use `linear_anon_page_index()` in `VM_SHARED` CoW fault code.



Have 24 - 28 bytes we free up to play with, so:

```
struct vm_area_struct {
        ...
        unsigned int vm_lock_seq;
        < 4 byte hole if per-VMA locks >
        struct list_head anon_vma_chain;
        struct anon_vma *anon_vma;
        ...
};
```

- Add new `vma->vm_anon_pgoff` field.
- Use `linear_anon_page_index()` on CoW.
- Use `vma->vm_pgoff` and `linear_page_index()` on any file operations.

Stabilisation: TODO

Being able to do everything under **RCU** is really **nice**. But also **terrible**.

Every point of **synchronisation** is gone and now we live in hell. How do we get back to sanity? 2 ideas:

1. **Mapping-granularity lock**
    a. **Can't allocate** memory for **every mapping**, defeats the object.
    b. **Can't lock** at **mm granularity**, defeats the object.
    c. Feasible: Introduce **ephemeral ranged locks** to get mapping granularity locking with far less overhead.
    d. Implementation: ??? profit (TBD)

2. **Re-engineer rmap walks to operate on page tables, tolerate races**
    a. **SLAB_TYPESAFE_BY_RCU** means we can avoid dangling VMA pointers, but fields may change.
    b. Really rmap walkers care about **page tables**, so need **RCU-safe** page table freeing.
    c. **Migration** a headache - needs stabilised mappings. Perhaps accumulate (mm, range) pairs per-folio?
    d. **Folio split** - serialises on rmap lock vs. collapses/other splits, would need to find another way.
    e. **MMU notifier** a nightmare - Can't do under RCU, and rmap walkers do it.
    f. Could **drop RCU** and stabilise mappings using an rmap walker **reference count** or similar.

Status/Discussion

- Have ~1.5kSLOC of *rough* PoC code (hence no RFC yet).
  - https://git.kernel.org/pub/scm/linux/kernel/git/ljs/linux.git/log/?h=project/cow-context
  - **Kernel boots**! But stabilisation TBD so... migration doesn't.
- Also TODO: KSM - tracks anon_vma objects, which won't exist any longer.
- Probably other things I haven't thought of...
- **Research** - this approach might not work out.
  - Even if it doesn't, it's useful to experiment!
  - I am intent on improving anon rmap, so will simply try a different approach if so.