Fine grained MM locking

Replacing mmap_sem with finer grained locks

Michel Lespinasse - Google

What is mmap_sem

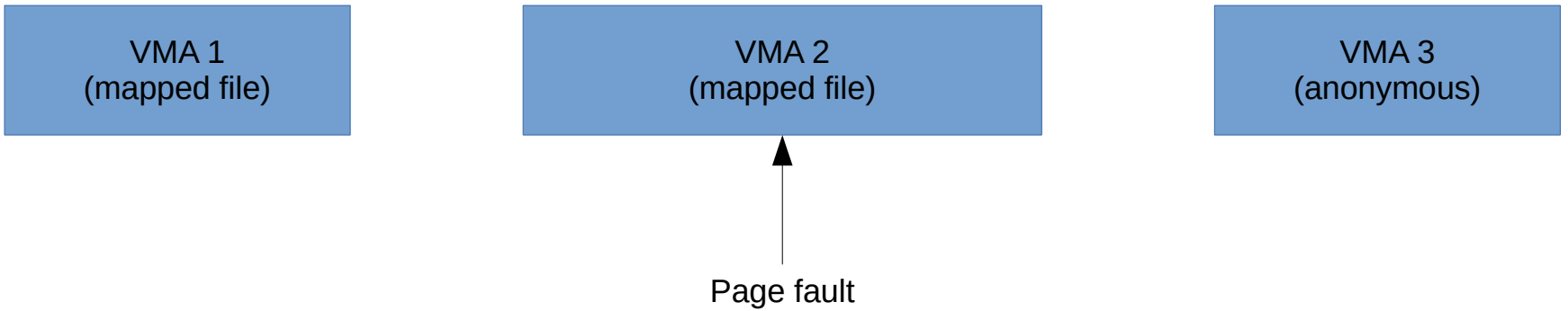
- RW semaphore
- Part of the MM structure
- Protects the VMA list / rbtree
  (and a ton of other per-MM things)
- Old design
  (and now kinda showing its age)

What is mmap_sem for

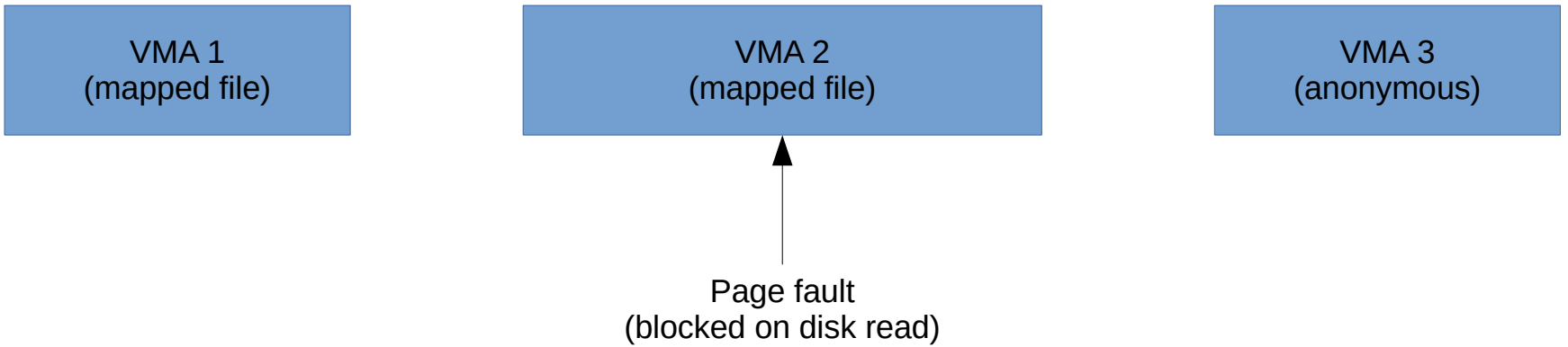
- Protects the VMA list / rbtree
  (and, again, a ton of other per-MM things)
- Prevents VMAs from being freed while another kernel thread is looking at them
- Prevents VMA mappings from changing while we run page faults (or other operations that depend on these mappings)

What is mmap_sem for

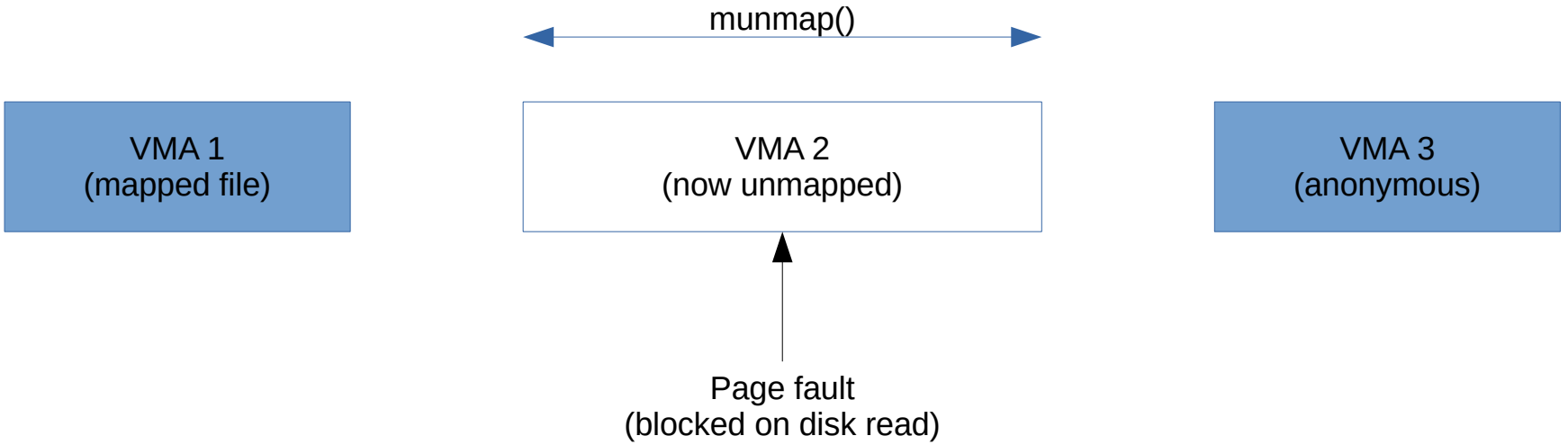
What is mmap_sem for



What is mmap_sem for



What is mmap_sem for



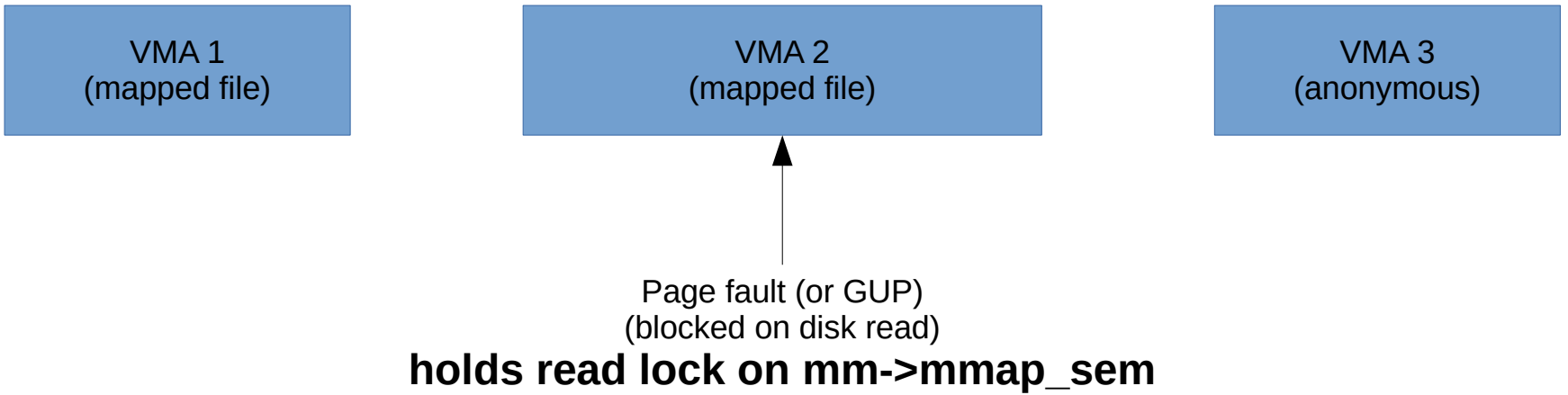
What is mmap_sem for

VMA 1
(mapped file)

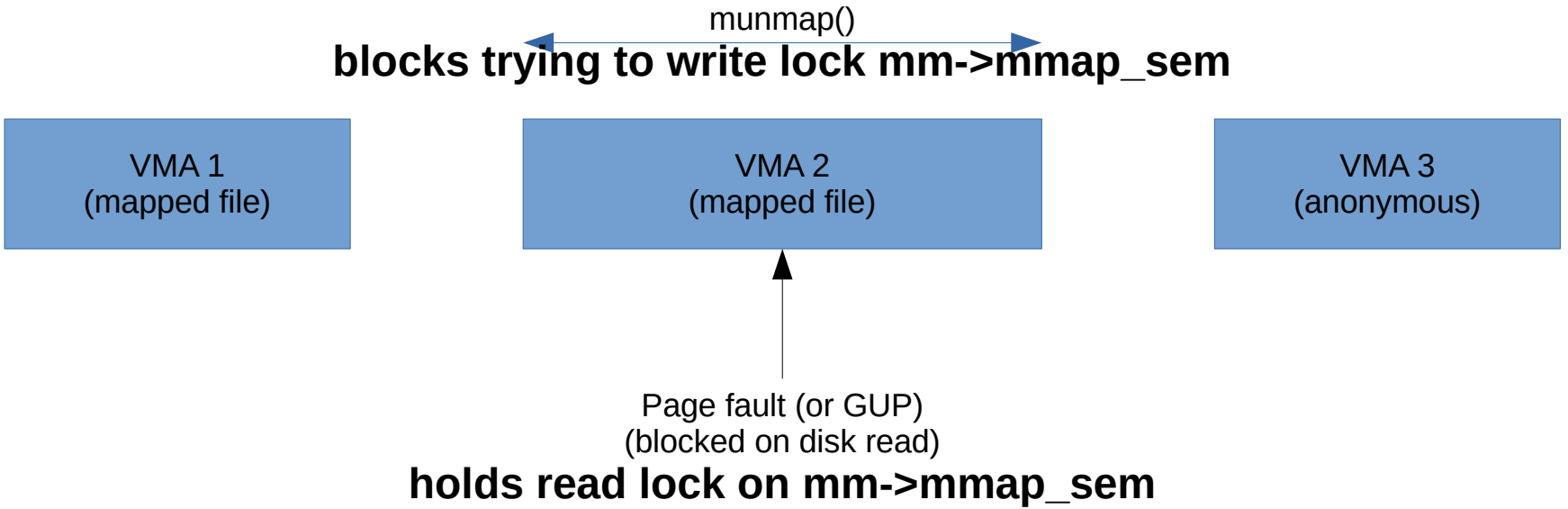
VMA 3
(anonymous)

Page fault
(disk read completed)
Now tries to map page into nonexistent VMA

What is mmap_sem for



What is mmap_sem for



mmap_sem is old!

- First added as a semaphore
  (equivalent to today's mutex)
  in linux v2.0.19 (September 1996)
- Converted to rw_semaphore
  in linux v2.4.3 (March 2001)
- No major design changes since,
  but a lot of band-aids / workarounds.

What changed ?

- Well, it’s been 20 years, so many things :)
- Larger memory size
  (some operations protected by mmap_sem take time proportional to the number of pages affected)
- Ubiquitous multi-threading
  (we actually care about multi-threading performance today; we mostly did not 20 years ago)
- Wider range of storage device speeds

The problem with mmap_sem

- The situation described earlier (true conflict between fault and munmap) is extremely uncommon !
- Correct threaded programs normally avoid having their threads race against each other
- Multiple threads may run non-overlapping memory operations (which are logically independent of each other), but the kernel does not make that distinction (thus causing false conflicts)

mmap_sem false conflicts

- Some threads allocate or free some (large) memory blocks
- Some threads access memory they have already allocated
- Some threads are getting spawned
  (thus requiring new user stack allocations)
- The sysadmin runs the ps command
  (which shows the process args stored in its address space)

mmap_sem mitigations

- FAULT_FLAG_ALLOW_RETRY (October 2010)
  Allows fault handler to release mmap_sem during known slow cases, such as hitting disk
- mm_populate() (January 2011)
  Only hold mmap_sem for read during mmap(MAP_POPULATE) and mlock; allow it to be released during known slow cases
- downgrade_write() in munmap() (October 2018)
  Only hold mmap_sem for read when zapping pages in munmap

mmap_sem mitigation limits

- Only handles the most common slow cases
  works in page faults hitting disk
  fails in get_user_pages()
  fails if the delay is caused by other reason (such as reclaim)
- Still confusing to application developers
  The mitigations only make the bad cases harder to hit
- Kernel code complexity
  The mitigations complicate mmap_sem locking… a lot.

What can we do about mmap_sem ?

- We have been painting ourselves into a corner trying to work around the limitations of the mmap_sem design (per-mm lock causing false conflicts)
- Can we redesign it to avoid the issue entirely ?

Goals for mmap_sem replacement

- Contention should ideally only occur between threads manipulating the same memory
- May block on locks protecting shared data structures, as long as they are only held for short amounts of time:
  - Not during file accesses,
  - Not while allocating user memory,
  - Not for operations that take O(pages) time

mmap_sem replacement strategy

- Make a working prototype
  Past discussions have often gotten stuck on details before the implementation stage. We need a working prototype, which can be evaluated on its own or as a basis for further improvements. (Perfect is the enemy of good)
- Progressive replacement
  Can not change all MM code at once:
  - Convert mmap_sem lockers one at a time
  - Convert vm_ops definitions one at a time

Supporting progressive replacement

- Our mmap_sem replacement needs to support both coarse lockers (automatically converted from the current mmap_sem uses) and fine grained lockers (with an associated address range to avoid false conflicts)
- Making a given locker fine grained does not change its interaction with coarse lockers.
  The only changing interaction is with other fine grained lockers.
  This greatly facilitates progressive conversion of each locker.

Converting one mmap_sem locker

- When the locker is converted to fine grained, it needs to protect against shared data structures being concurrently accessed by another fine grained locker
- Add new locks as necessary
- Ensure they are never held for long
  Not during file accesses,
  Not while allocating user memory,
  Not for operations that take O(pages) time.

Locking granularity

- Implementation choice: lock arbitrary address ranges, independently of the existing VMAs
- Supports the goal of avoiding false conflicts
- Some simplicity to it
  locking based on existing VMAs seems difficult, as the VMAs could change while we are waiting to acquire the locks
- Open to changing this if there was a strong proven performance justification

Basic implementation ideas

- Use a range locking data structure to represent current and pending range locks
- Address ranges may be locked for read or write
- Add new lock (mm→vma_lock) protecting both the VMA rbtree and the range locking structure
  (it can be convenient to know the VMA contents before acquiring a range lock...)

Putting it all together

(Some patchset I have been working on...)

MM locking API

- Add MM locking API
  (Initially implemented as rw_semaphore wrappers)
- Convert the existing mmap_sem call sites to the new API (mostly automated by coccinelle)
- Add fine grained range locking to the API
  (now implemented using an interval tree)
  At this point all lockers are still using coarse locks

First writer: do_mmap()

- Adapt do_mmap() API so it can take the MM lock on its own rather than relying on the caller
- vm_mmap_pgoff() calls do_mmap() and lets it take a write lock on a right sized MM range
- Fine grained in easiest cases
  (known address, anonymous memory, nothing to munmap)
  (double check nothing to munmap after acquiring range lock)
- Fall back to coarse in other cases

do_mmap() with unknown address

- get_unmapped_area()
- Mark the address range as pending allocation
- Acquire fine grained lock on the desired range
- Verify address range still marked as pending
- If not, release fine grained lock, acquire coarse lock, and retry get_unmapped_area().

do_mmap() with a file

- Not implemented yet (fall back to coarse lock)
- Some drivers expect a coarse lock when their file mmap() method is called
- Need to whitelist drivers

do_mmap() with existing VMAs

- Not implemented yet (fall back to coarse lock)
- Some drivers may expect a coarse lock when their vm_ops close() method is called
- Need to whitelist drivers
- Must make sure to release mm->vma_lock while zapping user pages

Page fault path: acquiring range lock

- Examine VMA for the faulting address
- Determine appropriate locking range for the address and VMA type
  (i.e., 2MB range around the address in the anon VMA case)
- Acquire read lock for the address range
- Verify VMA type has not changed
  Retry with coarse lock if it did.

Page fault path: faulting the page

- Fault the page as usual, based on VMA attributes obtained at the start of the fault. The attributes won’t change as the range is read locked.
- Note that VMA attributes have to be looked up at the start of the fault; the VMA can not be referenced later on as a fine grained writer may free it due to VMA merging.

Patch set status

- Still working on it.
- I want to share it soon; but I must finish the page fault path first for it to be meaningful.
- I hope having a concrete implementation will help test various ideas and foster interest in solving the mmap_sem false sharing issue.

Future plans

- Grow the number of places we do fine grained locking
- Performance comparison
- Expect we may have to add speculative faults to bring the performance up
- Possibly replace the centralized range lock with a distributed approach ??? If someone wants to tackle this… (not me)