Update on the SLUB allocator sheaves

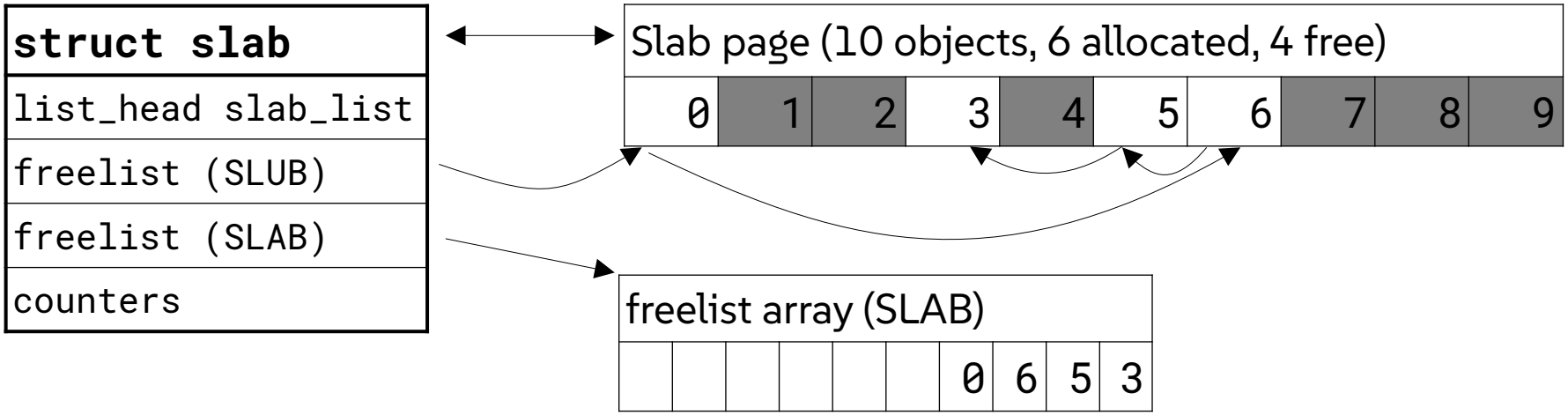
Vlastimil Babka, SUSE Labs
FOSDEM 2026, Kernel track

SUSE

SLUB percpu sheaves – initial motivation (2023/2024)

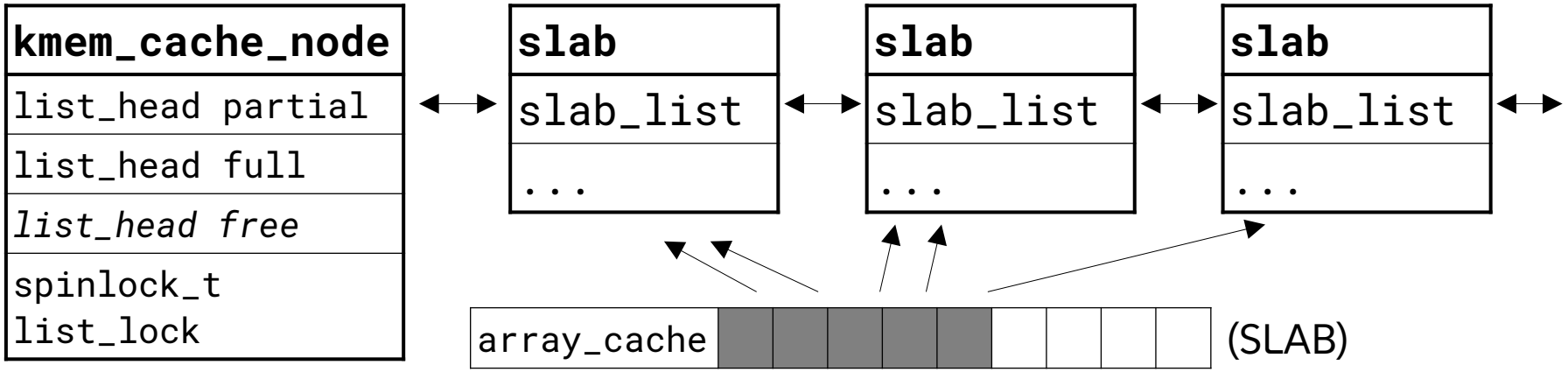
- What's the next step after removing SLOB and SLAB, leaving only SLUB?
- Overall goal: stop building caching layers on top of slab allocator (usually arrays of object pointers)
  - Some seem to do it just for performance reasons (block layer)
  - BPF: allocate/free in restricted contexts (e.g. NMI handler) not supported by slab
  - maple tree nodes: preallocate before an operation that can't fail in the middle, return unused objects afterwards
- Why? Code duplication, maintenance overhead, wasted memory, in some cases cpu overhead
  - Cached objects out of slab allocator's control, cannot be consolidated or reused
- Idea: once the allocator provides what users need, building layers on top is unnecessary
  - An array of object pointers in some form, but under the allocator's control
- Maybe also improve performance, combine only the good parts of SLUB and SLAB design?

SLUB (vs SLAB) basic functionality – single slab



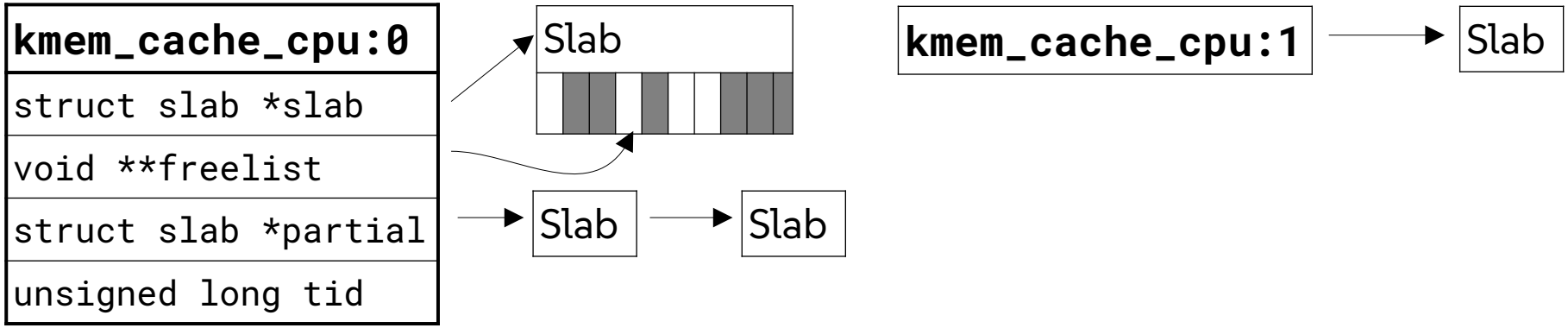
- SLUB has freelist embedded in free objects – single-linked, NULL-terminated
  - Allows allocating all objects in a single operation, multiple freelists in a single slab
  - Allows freeing to the slab with an atomic update of `freelist+counters` by the `try_cmpxchg128()` operation

SLUB (vs SLAB) – multiple slabs, SLAB percpu caching



- List of slabs per NUMA node with a spinlock – would not scale, needs some form of percpu caching
  - Turns out the caching is much easier to implement for allocating than for freeing
- SLAB - array caches: `percpu` (only NUMA local objects), `shared` (victim cache, same node), `alien` (freeing non-local objects, needed periodic flushing) – decided based on object's NUMA node provenance
  - Part of the array flushed to slabs (or `shared`) when full, part refilled from them when empty

SLUB percpu caching



- Allocations – fast from the percpu slab/freelist if available (and from a matching NUMA node)
  - `this_cpu_try_cmpxchg128()` of `freelist+tid` to avoid irq or preempt disable
- Freeing – fastpath similar to allocation, but actually unlikely, the object might be from another slab
  - Freeing to another slab is much more likely than freeing to remote NUMA node in SLAB
  - Slowpath is `try_cmpxchg128()` (atomic), sometimes also spinlock for the partial list

SLUB percpu sheaves – basics (v8) (LWN)

- Sheaf: fixed-capacity (per kmem_cache) array of object pointers from slabs
  - each cpu has two - `main` and `spare` sheaves, so that flushing or refilling is not partial
  - per-node `barn` is a victim cache for empty and full sheaves (up to `10`)
- Issues of SLAB design avoided by bypassing sheaves when freeing remote objects
  - Quick tracing session suggested it's only done in ~5% of the cases
  - No `alien` arrays that would need periodic flushing
- Percpu sheaves locking overhead limited by the `local_trylock_t` primitive
  - Uses cheaper preempt disable instead of irq disable in exchange for (rare) trylock failures
- Initially opt-in, capacity given when creating the kmem_cache
- Thanks to Matthew Wilcox for the terminology

SLUB percpu sheaves – handing out prefilled sheaves

- Support a usage pattern where a number of objects might be needed in a critical section that cannot block (and thus free memory to allocate) and also can't handle allocation failures
  - Maple tree nodes – exact number of allocations not known beforehand, only the worst case scenario can be pre-calculated
  - Previously handled by preallocating for the worst case, freeing back most of the preallocated objects afterwards – ineffective even with bulk alloc/free
- `kmem_cache_prefill_sheaf(size, gfp)` - ideally removes the `spare` sheaf, refills it only when it has less than `size` objects, returns it to the caller
- `kmem_cache_alloc_from_sheaf()` - guaranteed to succeed until depleted
- `kmem_cache_return_sheaf()` - ideally restores the sheaf as `spare` as-is, avoids flushing if possible

SLUB percpu sheaves – `kfree_rcu()` batching

- The existing `kfree_rcu()` implementation frees objects back to their slabs after the grace period
  - Batching exists to make it more efficient than a straightforward implementation
- Problem: freeing to percpu sheaves of many accumulated objects was overloading them
  - With `rcu_nocbs`, this freeing was restricted to some cpus only, others not refilled
- Solution: each cpu has additional `rcu_free` sheaf
  - `kfree_rcu()` adds to this sheaf; when full, `call_rcu()` called on the full sheaf
  - `rcu_free_sheaf()` callback will try to put the sheaf to barn for reuse

SLUB sheaves – current state and plans

- Sheaves were merged in 6.18 (which is a LTS kernel...)
- Opt-in, enabled only for VMA's and maple tree nodes
    - Both utilize `kfree_rcu()`
    - Maple nodes also use the prefilled sheaves API
- The kernel didn't explode :)

SLUB sheaves – current state and plans

- Being opt-in is far from ideal though!
  - It was OK as the first step to validate the implementation, fix bugs...
- Non-trivial implementation and maintenance overhead for only few users
  - Still backed by another caching layer with arguably even less trivial implementation
- How to determine which caches should benefit from the opt-in?
  - What sheaf size? Should the size be fixed? Automatically selected?
- Thus, enabling sheaves for all caches is the next step
  - And then removing the existing caching based on cpu (partial) slabs

SLUB sheaves – enabling for all caches

- Series (v4, 22 patches), now in linux-next for several weeks; thanks to all the reviewers!
- Some fun with bootstrapping and recursion prevention
  - Sheaves are allocated by `kmalloc()`, initial percpu sheaves have to be instantiated at cache creation, so how to create caches for kmalloc without allocating sheaves?
  - Allocating from a `kmalloc()` cache might result in allocating a sheaf, which can (depending on the size) be actually allocated from the very same cache?
  - Both solvable and not completely new problems in slab allocators' history
- Sheaf capacity calculated automatically to roughly match how many objects the cpu (partial) slabs would cache on average - subject to possible future tuning
  - Also rounded up to nearest `kmalloc()` object size
  - Any explicitly specified size now treated as a minimum

SLUB sheaves – enabling for all caches

- Lots of code removed afterwards :) including usage of `this_cpu_try_cmpxchg128()`
  - `try_cmpxchg128()` usage remains to put free objects back to their slab's freelist
- Lots of fun with a memory leak due to missing "`break;`" in an earlier version
  - Not found by Chris Mason's AI powered review until retrospectively told to look for leaks
  - After his prompts adjustments, now should be found in 5 of 6 cases

So what about performance?

- Sorry, won't be presenting fancy graphs here :)
- The initial implementation was validated on in-kernel microbenchmarks
  - Also Suren Baghdasaryan had measured `mmap()` operation improvements
- For enabling sheaves, on all caches, there were benchmarks done by Harry Yoo and Petr Tesařík
  - In general showing an improvement, but with specific cases showing worse performance
  - But that was older versions of the patches, e.g. with the memory leak still present
- Due to being such a low-level implementation detail, it appears difficult to benchmark reliably
  - Things like subtly different memory layout may play a role due to processor caches
- Kernel test robot reports – some regressions to investigate, including workloads known to be sensitive to various corner cases, so should not be show-stoppers on their own
- Many performance aspects are the matter of tuning – sheaf sizes, barn sizes, and various heuristics

What are the tradeoffs?

- Thanks to Christoph Lameter for summarizing them
- CPU slabs (before sheaves) – consecutive allocations served from the same CPU slab
  - Spatial locality, good for TLB overhead
  - Worse for cache performance (objects might be cold)
- Sheaves – allocations are typically returning recently freed objects (LIFO)
  - Temporal locality, good for cache performance (objects might still be hot)
  - Potentially scattered from many slabs (worse for TLB)
  - Might also increase memory overhead (testing so far suggests it's not significant)
- Object freeing – fastpath is more likely with sheaves, and a bit faster thanks to `local_trylock_t`
  - Slowpath is identical – `try_cmpxchg128()`

Final Thoughts

- Implementation-wise – sheaves less tricky than `this_cpu_try_cmpxchg128()` based fastpaths
- More suitable for `kmalloc_nolock()` - a variant for alloc/free in restricted context such as NMI
  - Replaces BPF object caches (one of the original motivation of sheaves)
    - Without being based on sheaves (but also merged in 6.18)
  - Much of its complexity and limitations (mainly due to `PREEMPT_RT`) are removed with CPU slabs replaced by sheaves
- A strong argument for completing the sheaves conversion, hence aiming for 7.0
- Future work – integrate better with existing `kfree_rcu()` batching – now mostly bypassed

SUSE

Thank you

Questions?

Effects on code size - merging of sheaves in 6.18

```
15 files changed, 2280 insertions(+), 1458 deletions(-)

include/linux/local_lock_internal.h |    9 +-
include/linux/maple_tree.h          |    6 +-
include/linux/slab.h                |   47 +++++
lib/maple_tree.c                    |  667 ++++++++++++--------------------------------------
--------------------
lib/test_maple_tree.c               |  137 ---------------
mm/slab.h                           |    5 +
mm/slab_common.c                    |   34 +++-
mm/slub.c                           | 1756 ++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++------
mm/vma_init.c                       |    1 +
tools/include/linux/slab.h          |  165 +++++++++++++++++-
tools/testing/radix-tree/maple.c    |  514 +++-----------------------------------------------
----
tools/testing/shared/linux.c        |  120 ++++++++++---
tools/testing/shared/maple-shared.h |   11 ++
tools/testing/shared/maple-shim.c   |    7 +
tools/testing/vma/vma_internal.h    |  259 +++++++--------------------
```

mm/ only: `5 files changed, 1790 insertions(+), 53 deletions(-)`

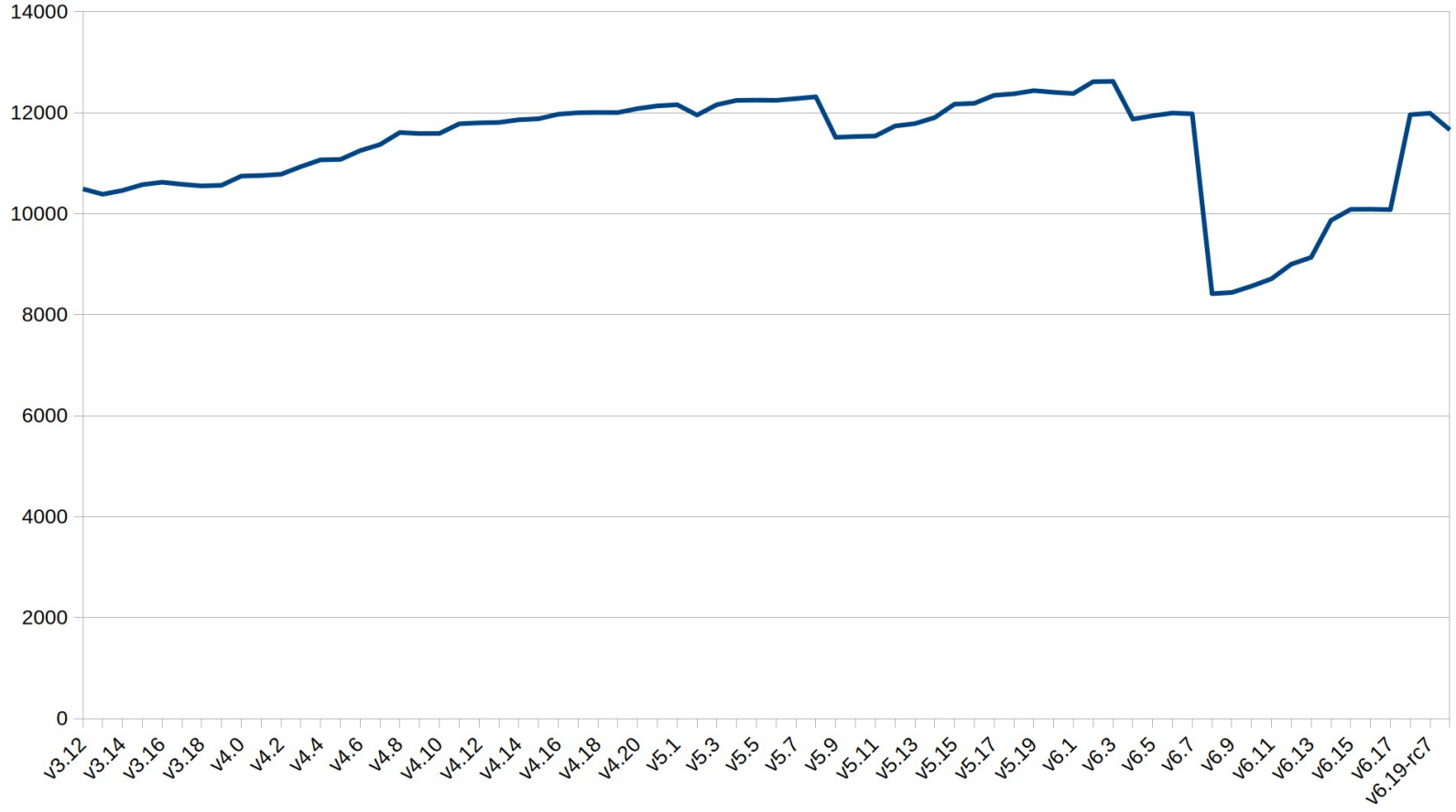
Effects on code size – replacing CPU slabs with sheaves

- Current diffstat of slab/for-7.0/sheaves

  - ```
    include/linux/slab.h |    6 -
    mm/Kconfig           |   11 -
    mm/internal.h        |    1 +
    mm/page_alloc.c      |    5 +
    mm/slab.h            |   57 ++---
    mm/slab_common.c     |    9 +-
    mm/slub.c            | 2663 +++++++++++++++++++++++++++++++++++++++++
    ++++++++++++++++++++++++++++++---------------------------------------
    ---------------------------------------------------------------------
    --------------------
    7 files changed, 978 insertions(+), 1774 deletions(-)
    ```

- Together with sheaves introduction: net addition of 941 lines (mm only) or 26 lines (all)
    ```

Evolution of slab allocator(s) code size



Evolution of personal diffstat (lines added minus removed)