Waste-Free Per-CPU Userspace Memory Allocation

Mathieu Desnoyers, EfficiOS

FOSDEM 2025
February 1 & 2, 2025
Brussels, Belgium

EfficiOS

Presentation Goals

- Discuss scaling of data structures by partitioning.
- Discuss challenges associated with use of per-CPU data in user-space: memory use, false sharing, cache line waste.
- Present the librseq mempool per-CPU allocator.
- Discuss current mmap(2)/madvise(2)/memfd_open(2) limitations with respect to shared mappings meant to be local to a process (mm).

Scaling Data Structures

- Scope of data structures,
- Partitioning data structures.

Scope of Data Structures

- Local variable (stack),
- Static definition (data),
- Dynamic allocation (heap).

Partitioning Data Structures

- Global variables
  - Single instance used across all threads/CPUs.
- Thread-Local Storage (TLS)
  - Each thread accesses its own data.
- Per-CPU data
  - Each CPU accesses its own data.

Thread-Local Storage (TLS)

- Inefficient use of CPU cache when the workload has more threads than the system has CPUs,
- Static definition only,
- Initialization of large TLS areas slows down thread creation,
- Global Dynamic TLS model for shared objects is slower than Initial Exec and have additional side-effects.

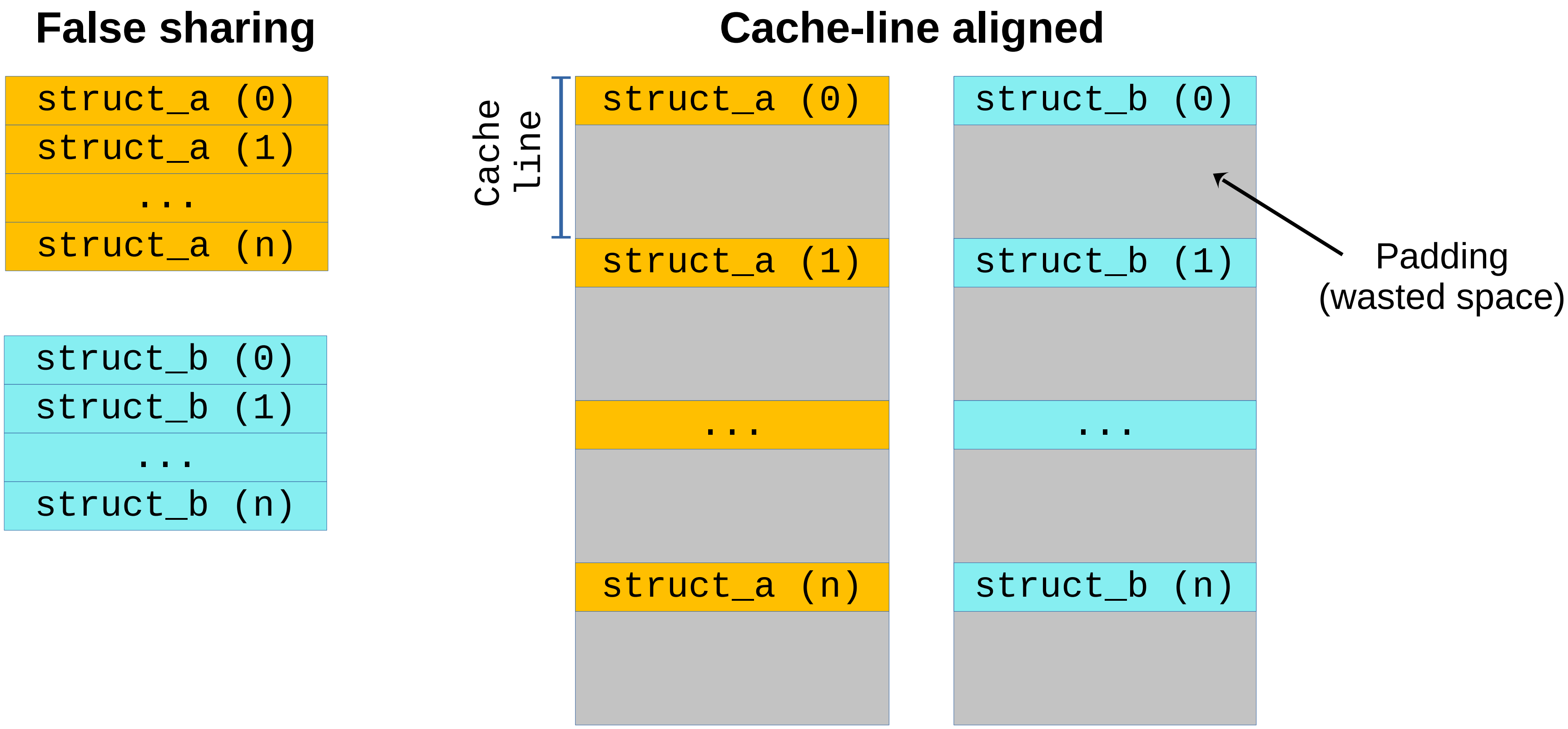
Per-CPU Data: An Alternative to TLS

- Partitioning strategy widely used within the Linux kernel,
- Not so much in user-space.

Anti-Pattern: Array of Per-CPU Items

- Array of N elements, N equals to number of possible CPUs,
- Index accesses with sched_getcpu(3), RSEQ cpu_id field, or
- Index accesses with RSEQ concurrency ID (mm_cid field) since Linux v6.3.

Anti-Pattern: Array of Per-CPU Items



Downsides of Per-CPU Array Anti-Pattern

- If elements **are not** cache-line aligned:
  - False sharing which hurts performance,
- If elements **are** cache-line aligned:
  - Waste precious cache line bytes due to padding,
  - Reduce functional density of CPU cache.

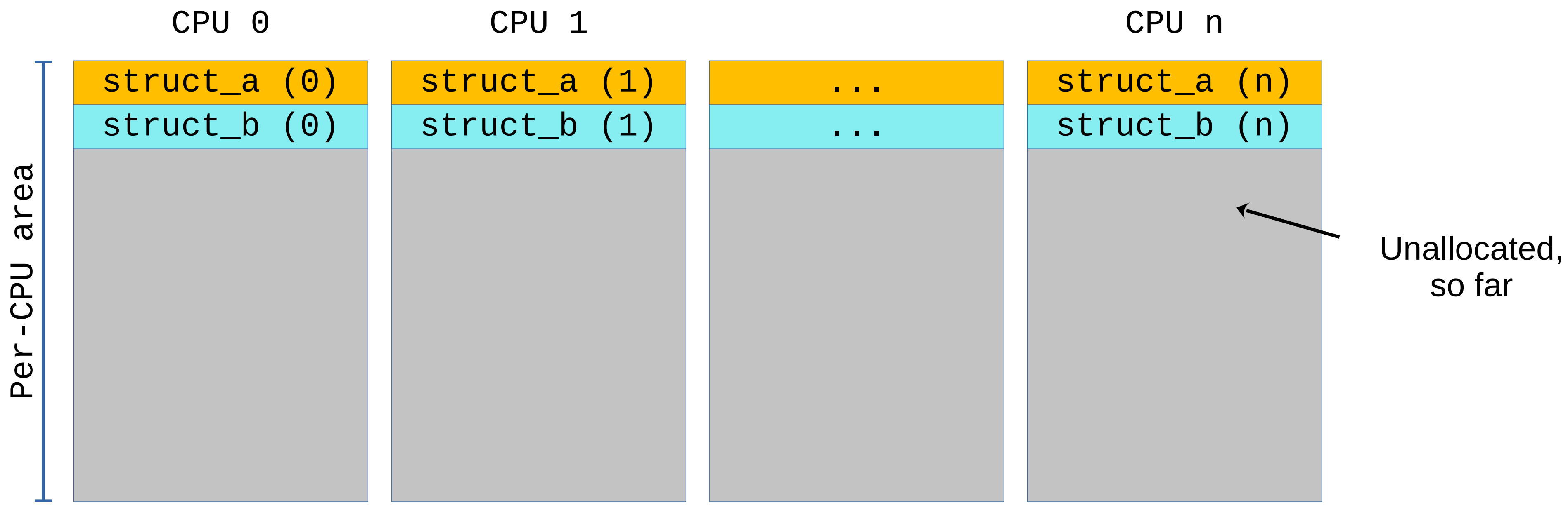
Linux Kernel Per-CPU Allocator

- Per-CPU memory allocator,
- Map a memory range on each CPU,
- The memory allocator allocates ranges at the same offset on each CPU.

Librseq Mempool Per-CPU Allocator

- Port of per-CPU Linux kernel allocator concepts to user-space,
- Implemented as a user-space API within librseq.
- Creation of memory pools. Each pool maps a memory range, which is an array of per-CPU areas (e.g. 64kB per CPU).
- Allocation against a pool reserves memory at same offset **for each CPU**.

Layout of Mempool Range



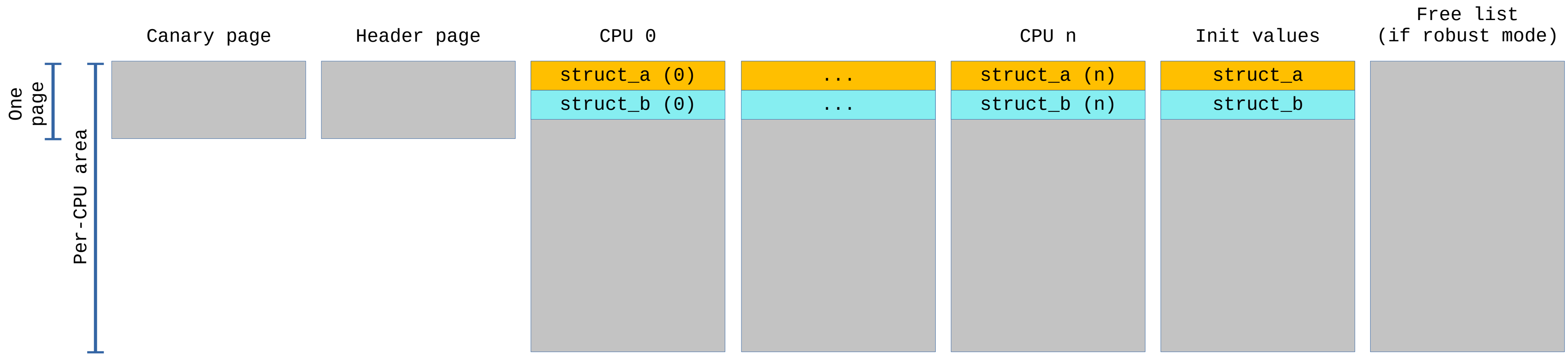
Mempool Access Pattern

- Replace array of per-CPU variables anti-pattern:
  - item_array_base_ptr + (cpu * sizeof(item))
- With range-stride based pattern:
  - item_ptr + (cpu * pool_stride)
  - Default pool stride: 64kB

Allocation From Pool

- Return a pointer in the area of CPU 0,
- Combines information about base of pool ranges and offset of item.

Mempool Range Layout with Metadata



Freeing Items From Pool

- Support for multiple pools provides isolation between users,
- Wish to do so without requiring the free API to take extra arguments besides pointer to free.

Reaching Pool Free-List From Pointer

- Map each pool range at aligned address,
- Find base by applying a mask
  - Similar to Linux kernel finding task struct from stack pointer,
- Header page before base of range.
  - Contains header structure describing range, pool, free-list.
- Aligned mmap(2) is not exposed by the Linux kernel.
- Implement it in userspace with mmap(2) of larger range, followed by unmap(2) of unused areas.

Memory Initialization

- Initializing newly allocated items by storing to each possible CPU memory area reserves a lot of resident memory on large systems,
- Systems with 512+ hardware threads are inreasingly common (e.g. AMD EPYC),
- Users restrict CPUs with cpusets or sched affinity.

Memory Initialization

```
void __rseq_percpu *
    rseq_mempool_percpu_malloc_init(struct rseq_mempool *pool,
                                    void *init_ptr, size_t init_len);
```

- Init-range shared mapping (memfd),
- Each CPU is a private copy-on-write (CoW) mapping of the init-range.
- CoW mappings only populate pages on store.

Memory Initialization

- Write initial content to the newly allocated area within the init-range,
- Iterate on all possible CPUs, read content visible from each CPU mapping, compare with init-range content,
- If it matches, no need to store to the per-CPU mapping,
- On mismatch, a CoW happened for the page due to stores from that CPU,
  - Need to store initial content to that CPU mapping.
- Ensures that memory is only reserved when actively used (stored to) by active CPUs.

fork(2)/clone(2)

- The init-range shared mapping is unfortunately shared across parent/children processes,
- Would ideally require introducing a new memfd anonymous file type private per-process
  - e.g. a new memfd_open(2) MFD_PRIVATE flag.
- Inconvenient work-around using madvise(2) MADV_DONTFORK to remove memory mappings from children processes and MADV_WIPEONFORK on canary page to allow detection of use across fork.

Additional Features Available

- Pool auto-expand with additional ranges when a range is fully allocated, up to a configurable upper bound.
- A mempool can be configured to either copy-on-write from init-range or from the zero page.
- Robust free list corruption checks (double-free, leaks on pool destruction, free-list corruption, poison values corruption).
- Mempool set, which is a collection of power-of-2 allocation size pools, allowing allocation of variable length data with a binning approach.

Future Work

- Add support for allocation of variable sized elements within a pool.
- Add a guard page between per-CPU data to eliminate cache line bouncing caused by hardware prefetch in sequential access patterns.
- Figure out a way to have an anonymous file private to a process (e.g. memfd MFD_PRIVATE). Meanwhile can work-around the single-threaded use-case with a copy in pthread_atfork() handler.
- Improve cgroup cpu controller to allow expressing concurrency limits without cpusets. This would facilitate limiting memory use of per-CPU data structures indexed by concurrency IDs within containers on machines with many CPUs.

memfd_create(2) MFD_PRIVATE

- Fork/clone can be handled more robustly by adding a MFD_PRIVATE flag to memfd_create(2):
  - Allow many shared mappings of the memfd anonymous file within a process,
  - Each process gets its own “private” anonymous file.
- Use-cases:
  - Mempool per-CPU allocator: init-range is a shared mapping, with CoW private mappings for per-CPU ranges,
  - Mesh allocator requires this as well. It also maps the same physical page at different addresses to reduce internal allocator fragmentation.
    - Mesh: Compacting Memory Management for C/C++ Applications
    - https://dl.acm.org/doi/pdf/10.1145/3314221.3314582
  - Google dynamic analysis tools require many MAP_SHARED mappings of a given page within a process, behaving as MAP_PRIVATE on fork.

Questions / Comments ?

- Links:
  - https://git.kernel.org/pub/scm/libs/librseq/librseq.git/tree/include/rseq/mempool.h
  - https://git.kernel.org/pub/scm/libs/librseq/librseq.git/tree/src/rseq-mempool.c