NAME

smr - safe memory reclamation for lock-free data structures

SYNOPSIS

#include <sys/smr.h>

smr_seq_t
smr_advance(*smr_t smr*);

smr_t
smr_create(*const char *name*);

void
smr_destroy(*smr_t smr*);

void
smr_enter(*smr_t smr*);

void
smr_exit(*smr_t smr*);

bool
smr_poll(*smr_t smr*, *smr_seq_t goal*, *bool wait*);

void
smr_synchronize(*smr_t smr*);

void
smr_wait(*smr_t smr*, *smr_seq_t goal*);

DESCRIPTION

Safe Memory Reclamation (SMR) is a facility which enables the implementation of memory-safe lock-free data structures. In typical usage, read accesses to an SMR-protected data structure, such as a hash table or tree, are performed in a "read section" consisting of code bracketed by **smr_enter**() and **smr_exit**() calls, while mutations of the data structure are serialized by a traditional mutex such as mutex(9). In contrast with reader-writer locks such as rwlock(9), rmlock(9), and sx(9), SMR allows readers and writers to access the data structure concurrently. Readers can always enter a read section immediately (**smr_enter**() never blocks), so mutations do not introduce read latency. Furthermore, **smr_enter**() and **smr_exit**() operate only on per-CPU data and thus avoid some of the performance problems inherent in the implementation of traditional reader-writer mutexes. SMR can therefore be a

useful building block for data structures which are accessed frequently but are only rarely modified.

Note that any SMR-protected data structure must be implemented carefully such that operations behave correctly in the absence of mutual exclusion between readers and writers. The data structure must be designed to be lock-free; SMR merely facilitates the implementation, for example by making it safe to follow dangling pointers and by helping avoid the ABA problem.

When shared accesses to and mutations of a data structure can proceed concurrently, writers must take care to ensure that any items removed from the structure are not freed and recycled while readers are accessing them in parallel. This requirement results in a two-phase approach to the removal of items: first, the item is unlinked such that all pointers to the item are removed from the structure, preventing any new readers from observing the item. Then, the writer waits until some mechanism guarantees that no existing readers are still accessing the item. At that point the memory for that item can be freed and reused safely. SMR provides this mechanism: readers may access a lock-free data structure in between calls to the **smr_enter**() and **smr_exit**() functions, which together create a read section, and the **smr_advance**(), **smr_poll**(), **smr_wait**(), and **smr_synchronize**() functions can be used to wait for threads in read sections to finish. All of these functions operate on a *smr_t* state block which holds both per-CPU and global state. Readers load global state and modify per-CPU state, while writers must scan all per-CPU states to detect active readers. SMR is designed to amortize this cost by batching to give acceptable performance in write-heavy workloads.

Readers

Threads enter a read section by calling **smr_enter**(). Read sections should be short, and many operations are not permitted while in a read section. Specifically, context switching is disabled, and thus readers may not acquire blocking mutexes such as mutex(9). Another consequence of this is that the thread is pinned to the current CPU for the duration of the read section. Furthermore, read sections may not be nested: it is incorrect to call **smr_enter**() with a given *smr_t* state block when already in a read section for that state block.

UMA Integration

To simplify the integration of SMR into consumers, the uma(9) kernel memory allocator provides some SMR-specified facilities. This eliminates a good deal of complexity from the implementation of consumers and automatically batches write operations.

In typical usage, a UMA zone (created with the UMA_ZONE_SMR flag or initialized with the **uma_zone_set_smr**() function) is coupled with a *smr_t* state block. To insert an item into an SMR-protected data structure, memory is allocated from the zone using the **uma_zalloc_smr**() function. Insertions and removals are serialized using traditional mutual exclusion and items are freed using the **uma_zfree_smr**() function. Read-only lookup operations are performed in SMR read sections. **uma_zfree_smr**() waits for all active readers which may be accessing the freed item to finish their read

sections before recycling that item's memory.

If the zone has an associated per-item destructor, it will be invoked at some point when no readers can be accessing a given item. The destructor can be used to release additional resources associated with the item. Note however that there is no guarantee that the destructor will be invoked in a bounded time period.

Writers

Consumers are expected to use SMR in conjunction with UMA and thus need only make use of the **smr_enter**() and **smr_exit**() functions, and the SMR helper macros defined in *sys/smr_types.h*. However, an introduction to the write-side interface of SMR can be useful.

Internally, SMR maintains a global 'write sequence' number. When entering a read section, **smr_enter**() loads a copy of the write sequence and stores it in per-CPU memory, hence 'observing' that value. To exit a read section, this per-CPU memory is overwritten with an invalid value, making the CPU inactive. Writers perform two operations: advancing the write sequence number, and polling all CPUs to see whether active readers have observed a given sequence number. These operations are performed by **smr_advance**() and **smr_poll**(), respectively, which do not require serialization between multiple writers.

After a writer unlinks an item from a data structure, it increments the write sequence number and tags the item with the new value returned by **smr_advance**(). Once all CPUs have observed the new value, the writer can use **smr_poll**() to deduce that no active readers have access to the unlinked item, and thus the item is safe to recycle. Because this pair of operations is relatively expensive, it is generally a good idea to amortize this cost by accumulating a collection of multiple unlinked items and tagging the entire batch with a target write sequence number.

smr_poll() is a non-blocking operation and returns true only if all active readers are guaranteed to have observed the target sequence number value. **smr_wait**() is a variant of **smr_poll**() which waits until all CPUs have observed the target sequence number value. **smr_synchronize**() combines **smr_advance**() with **smr_wait**() to wait for all CPUs to observe a new write sequence number. This is an expensive operation and should only be used if polling cannot be deferred in some way.

Memory Ordering

The **smr_enter**() function has acquire semantics, and the **smr_exit**() function has release semantics. The **smr_advance**(), **smr_poll**(), **smr_wait**(), and **smr_synchronize**() functions should not be assumed to have any guarantees with respect to memory ordering. In practice, some of these functions have stronger ordering semantics than is stated here, but this is specific to the implementation and should not be relied upon. See atomic(9) for more details.

NOTES

Outside of FreeBSD the acronym SMR typically refers to a family of algorithms which enable memory-safe read-only access to a data structure concurrent with modifications to that data structure. Here, SMR refers to a particular algorithm belonging to this family, as well as its implementation in FreeBSD. See *sys/sys/smr.h* and *sys/kern/subr_smr.c* in the FreeBSD source tree for further details on the algorithm and the context.

The acronym SMR is also used to mean "shingled magnetic recording", a technology used to store data on hard disk drives which requires operating system support. These two uses of the acronym are unrelated.

SEE ALSO

atomic(9), locking(9), uma(9)

AUTHORS

The SMR algorithm and its implementation were provided by Jeff Roberson <*jeff@FreeBSD.org*>.
This manual page was written by
Mark Johnston <*markj@FreeBSD.org*>.