NAME

epoch, **epoch_context**, **epoch_alloc**, **epoch_free**, **epoch_enter**, **epoch_exit**, **epoch_wait**, **epoch_enter_preempt**, **epoch_exit_preempt**, **epoch_wait_preempt**, **epoch_call**, **epoch_drain_callbacks**, **in_epoch**, **in_epoch_verbose** - kernel epoch based reclamation

SYNOPSIS

```
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/epoch.h>

struct epoch;                   /* Opaque */
typedef struct epoch *epoch_t;

struct epoch_context {
        void    *data[2];
};
typedef struct epoch_context *epoch_context_t;
typedef void epoch_callback_t(epoch_context_t);

struct epoch_tracker;           /* Opaque */
typedef struct epoch_tracker *epoch_tracker_t;

epoch_t
epoch_alloc(const char *name, int flags);

void
epoch_free(epoch_t epoch);

void
epoch_enter(epoch_t epoch);

void
epoch_exit(epoch_t epoch);

void
epoch_wait(epoch_t epoch);

void
epoch_enter_preempt(epoch_t epoch, epoch_tracker_t et);
```

void
epoch_exit_preempt(*epoch_t epoch, epoch_tracker_t et*);

void
epoch_wait_preempt(*epoch_t epoch*);

void
epoch_call(*epoch_t epoch, epoch_callback_t callback, epoch_context_t ctx*);

void
epoch_drain_callbacks(*epoch_t epoch*);

int
in_epoch(*epoch_t epoch*);

int
in_epoch_verbose(*epoch_t epoch, int dump_onfail*);

DESCRIPTION

Epochs are used to guarantee liveness and immutability of data by deferring reclamation and mutation until a grace period has elapsed. Epochs do not have any lock ordering issues. Entering and leaving an epoch section will never block.

Epochs are allocated with **epoch_alloc**(). The *name* argument is used for debugging convenience when the **EPOCH_TRACE** kernel option is configured. By default, epochs do not allow preemption during sections. By default mutexes cannot be held across **epoch_wait_preempt**(). The *flags* specified are formed by *OR*'ing the following values:

EPOCH_LOCKED
: Permit holding mutexes across **epoch_wait_preempt**() (requires EPOCH_PREEMPT). When doing this one must be cautious of creating a situation where a deadlock is possible.

EPOCH_PREEMPT
: The *epoch* will allow preemption during sections. Only non-sleepable locks may be acquired during a preemptible epoch. The functions **epoch_enter_preempt**(), **epoch_exit_preempt**(), and **epoch_wait_preempt**() must be used in place of **epoch_enter**(), **epoch_exit**(), and **epoch_wait**(), respectively.

*epoch*s are freed with **epoch_free**().

Threads indicate the start of an epoch critical section by calling **epoch_enter**() (or **epoch_enter_preempt**() for preemptible epochs). Threads call **epoch_exit**() (or **epoch_exit_preempt**() for preemptible epochs) to indicate the end of a critical section. *struct epoch_tracker*s are stack objects whose pointers are passed to **epoch_enter_preempt**() and **epoch_exit_preempt**() (much like *struct rm_priotracker*).

Threads can defer work until a grace period has expired since any thread has entered the epoch either synchronously or asynchronously. **epoch_call**() defers work asynchronously by invoking the provided *callback* at a later time. **epoch_wait**() (or **epoch_wait_preempt**()) blocks the current thread until the grace period has expired and the work can be done safely.

Default, non-preemptible epoch wait (**epoch_wait**()) is guaranteed to have much shorter completion times relative to preemptible epoch wait (**epoch_wait_preempt**()). (In the default type, none of the threads in an epoch section will be preempted before completing its section.)

INVARIANTS can assert that a thread is in an epoch by using **in_epoch**(). **in_epoch**(*epoch*) is equivalent to invoking **in_epoch_verbose**(*epoch, 0*). If **EPOCH_TRACE** is enabled, **in_epoch_verbose**(*epoch, 1*) provides additional verbose debugging information.

The epoch API currently does not support sleeping in epoch_preempt sections. A caller should never call **epoch_wait**() in the middle of an epoch section for the same epoch as this will lead to a deadlock.

The **epoch_drain_callbacks**() function is used to drain all pending callbacks which have been invoked by prior **epoch_call**() function calls on the same epoch. This function is useful when there are shared memory structure(s) referred to by the epoch callback(s) which are not refcounted and are rarely freed. The typical place for calling this function is right before freeing or invalidating the shared resource(s) used by the epoch callback(s). This function can sleep and is not optimized for performance.

RETURN VALUES

in_epoch(*curepoch*) will return 1 if curthread is in curepoch, 0 otherwise.

EXAMPLES

Async free example: Thread 1:

```
int
in_pcbladdr(struct inpcb *inp, struct in_addr *faddr, struct in_laddr *laddr,
    struct ucred *cred)
{
    /* ... */
    epoch_enter(net_epoch);
```

```
    CK_STAILQ_FOREACH(ifa, &ifp->if_addrhead, ifa_link) {
        sa = ifa->ifa_addr;
            if (sa->sa_family != AF_INET)
                continue;
            sin = (struct sockaddr_in *)sa;
            if (prison_check_ip4(cred, &sin->sin_addr) == 0) {
                ia = (struct in_ifaddr *)ifa;
                break;
            }
    }
    epoch_exit(net_epoch);
    /* ... */
}
Thread 2:

void
ifa_free(struct ifaddr *ifa)
{

    if (refcount_release(&ifa->ifa_refcnt))
        epoch_call(net_epoch, ifa_destroy, &ifa->ifa_epoch_ctx);
}

void
if_purgeaddrs(struct ifnet *ifp)
{

    /* .... *
    IF_ADDR_WLOCK(ifp);
    CK_STAILQ_REMOVE(&ifp->if_addrhead, ifa, ifaddr, ifa_link);
    IF_ADDR_WUNLOCK(ifp);
    ifa_free(ifa);
}
```

Thread 1 traverses the ifaddr list in an epoch. Thread 2 unlinks with the corresponding epoch safe macro, marks as logically free, and then defers deletion. More general mutation or a synchronous free would have to follow a call to **epoch_wait**().

NOTES

The **epoch** kernel programming interface is under development and is subject to change.

SEE ALSO

callout(9), locking(9), mtx_pool(9), mutex(9), rwlock(9), sema(9), sleep(9), sx(9)

HISTORY

The **epoch** framework first appeared in FreeBSD 11.0.

CAVEATS

One must be cautious when using **epoch_wait_preempt**(). Threads are pinned during epoch sections, so if a thread in a section is then preempted by a higher priority compute bound thread on that CPU, it can be prevented from leaving the section indefinitely.

Epochs are not a straight replacement for read locks. Callers must use safe list and tailq traversal routines in an epoch (see ck_queue). When modifying a list referenced from an epoch section safe removal routines must be used and the caller can no longer modify a list entry in place. An item to be modified must be handled with copy on write and frees must be deferred until after a grace period has elapsed.