NAME

cradix — compact compressed radix tree with scalably parallelizable lookup

SYNOPSIS

#include <cradix.h>

struct cradix;
struct cradix_node;
struct cradix_ops;

void
cradix_init(*struct cradix *C*, *const struct cradix_ops *ops*);

void
cradix_destroy(*struct cradix *C*);

*struct cradix_node **
cradix_lookup(*const struct cradix *C*, *const void *key*, *size_t keylen*);

bool
cradix_insert_begin(*struct cradix *C*, *struct cradix_insert *I*);

void
cradix_insert_end(*struct cradix *C*, *struct cradix_insert *I*);

*struct cradix_node **
cradix_insert(*struct cradix *C*, *struct cradix_node *N*,
    *struct cradix_insert *I*);

void
cradix_delete_begin(*struct cradix *C*, *struct cradix_delete *D*);

void
cradix_delete_end(*struct cradix *C*, *struct cradix_delete *D*);

void
cradix_delete(*struct cradix *C*, *struct cradix_node *N*,
    *struct cradix_delete *D*);

*struct cradix_node **
cradix_delete_key(*struct cradix *C*, *uint64_t key*, *struct cradix_delete *D*);

DESCRIPTION

The **cradix** library implements compact compressed radix trees with scalably parallelizable lookup.

radix tree
: Tree-structured set of elements with string keys in lexicographic order, compressed to elide single-child branches. Time to lookup/insert/delete is O(1), since the maximum key length is bounded by 64 bits. The radix is 16 or 32 depending on compile options.

compressed
: Every branch always has at least two children, and many of them have more children.

compact
: Branches are allocated only as large as they need to be — if a branch has only 2 children, it won't use space for 32 children except if it had 32 children before and many of them were deleted without a subsequent insertion happening at the branch.

scalably parallelizable lookup
: With pserialize(9) or similar, multiple calls to **cradix_lookup**() and other lookup operations can safely run in parallel with each other and with insertion and deletion with no interprocessor synchronization in the lookup operations.

However, insertion and deletion must be serialized.

The caller must allocate storage for a *struct cradix* object and one *struct cradix_node* object for every element in the cradix tree. These objects should otherwise be treated as opaque, and not inspected, modified, or copied. Each element has a corresponding *key*, which is 64-bit integer.

The caller must define operations collected in a *struct cradix_ops* object to retrieve the key of an element, and optionally to allocate, free, and 'recycle' memory. 'Recycling' here means waiting until all readers have finished using the memory before freeing it.

```
struct cradix_ops {
        const uint64_t  (*key)(const struct cradix *C,
                            const struct cradix_node *N);
        void *          (*alloc)(const struct cradix *C, size_t size);
        void            (*free)(const struct cradix *C, void *ptr,
                            size_t size);
        void            (*recycle)(const struct cradix *C, void *ptr,
                            size_t size);
};
```

The operations are:

key(*C*, *N*)
Given a cradix tree *C* and a node *N*, return the node's key.

alloc(*C*, *size*)
Allocate an object *size* bytes long and return it, or return NULL on failure.

free(*C*, *ptr*, *size*)
Free *ptr*, an object *size* bytes long returned by **alloc**(). The object is gurantee not to be currently in use.

recycle(*C*, *ptr*, *size*)
Wait until all readers of *C* have finished, and then free *ptr*, an object *size* bytes long returned by **alloc**(). The object may currently be in use by readers, so the caller must wait for readers to finish with pserialize_perform(9) or similar.

INITIALIZATION AND FINALIZATION

cradix_init(*C*, *ops*)
Initialize a cradix tree with the specified operations. Must be done before any other operations on *C*.

cradix_destroy(*C*)
Destroy a cradix tree. The tree must be empty.

This operation is not necessary, but it provides a debugging aid by asserting that the tree is in a consistent state and preventing subsequent use.

LOOKUP/INSERT/DELETE

The following lookup operations may be done in parallel with any other operations. The lookup operations issue memory operations in an order necessary to synchronize with parallel modification operations.

cradix_lookup(*C*, *key*)
Return a pointer to the *struct cradix_node* object of the element whose key is *key*, or NULL if there is no such element.

cradix_insert_begin(*C*, *I*)
Preallocate enough memory to guarantee that a subsequent **cradix_insert_begin**() operation will succeed. Returns true if successful, false if unable to allocate enough memory to guarantee suc-

cess of **cradix_insert_begin**(). Must be followed by **cradix_insert_end**().

cradix_insert_end(*C*, *I*)
Free any unused memory allocated by **cradix_insert_begin**() after **cradix_insert**().

cradix_insert(*C*, *N*, *I*)
Insert the element whose *struct cradix_node* object is *N* into the cradix tree *C*. If there was already such an element in *C*, return a pointer to its *struct cradix_node* object and leave the cradix tree alone. Otherwise, return *N*. Thus, **cradix_insert**() always returns the element that is actually in *C*.

If *I* is nonnull, it must have been initialized by **cradix_insert_begin**(), and **cradix_insert**() will use memory preallocated in *I*, and will never fail. Otherwise, if *I* is `NULL`, then **cradix_insert**() may call the **alloc**() operation to allocate a branch structure if necessary; if that fails, then **cradix_insert**() will return `NULL` to indicate memory allocation failure.

cradix_delete_begin(*C*, *D*)
Prepare for deletion. Must be followed by **cradix_delete_end**().

cradix_delete_end(*C*, *D*)
Recycle any memory no longer needed after a deletion by calling the **recycle**() operation.

cradix_delete(*C*, *N*, *D*)
Delete the element whose *struct cradix_node* object is *N* from the cradix tree *C*. Caller must guarantee that *N* is currently in *C*.

If *D* is nonnull, it must have been initialized by **cradix_delete_begin**(), and **cradix_delete**() stashes any memory no longer used by *C* in *D* so that a subsequent **cradix_delete_end**() can recycle it. Otherwise, if *I* is `NULL`, then **cradix_delete**() may call the **recycle**() operation to recycle a branch structure if necessary.

cradix_delete_key(*C*, *key*)
Delete the element whose key is *key* from the cradix tree *C* and return a pointer to its *struct cradix_node* object, if there is one. If there is no such element, return `NULL` instead.

If *D* is nonnull, it must have been initialized by **cradix_delete_begin**(*C*, *D*), and **cradix_delete**(*C*, *key*, *D*) stashes any memory no longer used by *C* in *D* so that a subsequent **cradix_delete_end**(*C*, *D*) can recycle it. Otherwise, **cradix_delete_key**(*C*, *key*) `NULL` may call the **recycle**() operation to recycle a branch structure if necessary.

EXAMPLES

Example application in the NetBSD kernel. Definitions:

```
struct widget {
        uint64_t                key;
        struct cradix_node      node;
        ...
};

struct {
        kmutex_t                lock;
        struct cradix           tree;
        struct pserialize       *psz;
} widgets __cacheline_aligned;

static uint64_t
widget_key(const struct cradix *C, const struct cradix_node *N)
```

```
{
        const struct widget *W = container_of(N, struct widget, node);

        KASSERT(C == &widgets.tree);
        return W->key;
}

static void
widget_recycle(struct cradix *C, void *ptr, size_t size)
{
        KASSERT(C == &widgets.tree);

        /* Wait for all pserialize read sections to complete. */
        pserialize_perform(widgets.psz);

        /* Memory is no longer in use; free it. */
        kmem_free(ptr, size);
}

static const struct cradix_ops widget_cradix_ops = {
        .key = &widget_key,
        .recycle = &widget_recycle,
};
```

Lookup:

```
        uint64_t key = ...;
        struct widget *W;
        int result = -1;
        int s;

        s = pserialize_read_enter();
        W = cradix_lookup(&widgets.tree, key);
        if (W == NULL)
                goto out;
        result = bromble(W);
out:    pserialize_read_exit(s);
```

Insert:

```
        uint64_t key = ...;
        struct widget *W, *W0;
        struct cradix_insert insert;

        W = alloc_widget();
        W->key = key;
        W->... = ...;

        if (!cradix_insert_begin(&widgets.tree, &insert)) {
                free_widget(W);
                return ENOMEM;
        }
        mutex_enter(&widgets.lock);
        W0 = cradix_insert(&widgets.tree, W, &insert);
```

```
mutex_exit(&widgets.lock);
cradix_insert_end(&widgets.tree, &insert);

if (W0 != W) {
        /* Failed to insert -- key already used. */
        free_widget(W);
        return EEXIST;
}
```

Delete:

```
uint64_t key = ...;
struct widget *W;
struct cradix_delete delete;

cradix_delete_begin(&widgets.tree, &delete);
mutex_enter(&widgets.lock);
W = cradix_delete_key(&widgets.tree, &key, sizeof key);
mutex_exit(&widgets.lock);
cradix_delete_end(&widgets.tree, &delete);

if (W != NULL)
        free_widget(W);
```

SEE ALSO

membar_ops(3), rbtree(3), pserialize(9)

Daniel J. Bernstein, *Crit-bit trees*, https://cr.yp.to/critbit.html.

AUTHORS

Taylor R Campbell ⟨campbell+cradix@mumble.net⟩

BUGS

Insertion and deletion must be serialized.