NAME

rsrc_m, rsrc_i – Resource Manager and Iterator Classes

SYNOPSIS

```
#include <rsrc.h>

template <class TYPE, class KEY>
class rsrc_m : public w_base_t {
    friend class rsrc_i<TYPE, KEY>;
public:
    NORET               rsrc_m(
      TYPE*                 space,
      int                   n,
      char*                 descriptor=0);
    NORET               ~rsrc_m();

    void                mutex_acquire();
    void                mutex_release();

    bool                is_cached(const KEY& k);

    w_rc_t              grab(
      TYPE*&                ret,
      const KEY&            k,
      bool&                 found,
      bool&                 is_new,
      latch_mode_t                mode = LATCH_EX,
      int                   timeout = sthread_base_t::WAIT_FOREVER);

    w_rc_t              find(
      TYPE*&                ret,
      const KEY&            k,
      latch_mode_t                mode = LATCH_EX,
      int                   ref_bit = 1,
      int                   timeout = sthread_base_t::WAIT_FOREVER);

    void                publish_partial(const TYPE* rsrc);
    void                publish(
      const TYPE*           rsrc,
      bool                  error_occurred = false);

    bool                is_mine(const TYPE* rsrc);

    void                pin(
      const TYPE*           rsrc,
      latch_mode_t                mode = LATCH_EX);

    void                upgrade_latch_if_not_block(
      const TYPE*           rsrc,
      bool&                 would_block);

    void                unpin(
      const TYPE*&                rsrc,
      int                   ref_bit = 1);
```

```
    // number of times pinned
    int                     pin_cnt(const TYPE* t);
    w_rc_t              remove(const TYPE*& t) {
      w_rc_t rc;
      bool get_mutex = ! _mutex.is_mine();
      if (get_mutex)    W_COERCE(_mutex.acquire());
      rc = _remove(t);
      if (get_mutex)    _mutex.release();
      return rc;
    }

    void                dump(ostream &o,bool debugging=1)const;
    int                     audit(bool prt= false) const;

    void                snapshot(u_int& npinned, u_int& nfree);

    unsigned long           ref_cnt, hit_cnt;

// iterator
template <class TYPE, class KEY>
class rsrc_i {
public:
    NORET               rsrc_i(
      rsrc_m<TYPE, KEY>&          r,
      latch_mode_t                m = LATCH_EX,
      int                   start = 0)
      : _mode(m), _idx(start), _curr(0), _r(r) {};

    NORET               ~rsrc_i();

    TYPE*               next();
    TYPE*               curr()      { return _curr ? _curr->ptr : 0; }
    w_rc_t              discard_curr();

private: // disabled methods
    NORET               rsrc_i(const rsrc_i&);
    rsrc_i&             operator=(const rsrc_i&);
};

/*
 *  rsrc_t
 *    control block (handle) to a resource
 */
template <class TYPE, class KEY>
struct rsrc_t {
public:
    NORET               rsrc_t()    {};
    NORET               ~rsrc_t()   {};
    w_link_t                link;       // used in resource hash table
    latch_t                 latch;            // latch on the resource
    KEY                     key;        // key of the resource
    KEY                     old_key;
    bool                old_key_valid;
```

```
        TYPE*               ptr;        // pointer to the resource
        w_base_t::uint4_t         waiters;    // # of waiters
        w_base_t::uint4_t         ref;        // ref count
        scond_t             exit_transit;     // signaled when
                                              // initialization is done

    };
```

DESCRIPTION

The **rsrc_m** template class manages a fixed size pool of "resources" (of type T) in a multi-threaded environment. A structure, **rsrc_t,** is associated with each resource. Class **rsrc_t** contains a key, K, a pointer to the resource and a latch to protect access to the resource. The **rsrc_t** elements are stored in a hash table, **hash_t.** Because of the latches, each resource can be individually "pinned" for any desired length of time without restricting access to other resources.

The template class **rsrc_i** is the iterator for the **rsrc_m** class.

When a entry needs to be added and the table is full, on old entry is removed based on an LRU policy.

The rsrc_m is relatively expensive, so it is probably best used to manage large resources or where high concurrency is needed. A good example is managing access to pages in a buffer pool.

Requirements:
The **rsrc_m** template takes two class parameters:

T the class type of the resources to be manages.

K the unique key of the resource for lookup purposes. *Note:* that **K** must define **K::operator=()** for copying since **rsrc_m** saves a copy of **K**for**lookup**purpose,**and u_long hash(const K&)** hash function for **K** because **rsrc_m** is hash-table based.

A resource in **rsrc_m**
can be in one of three states:

unused the resource is free; no key is associated with the resource.

cached the resource is cached and is associated with a key.

in-transit
the resource is begin replaced; its key is being changed.

Rsrc_m Interface

rsrc_m(rsrc, cnt, desc)

The constructor creates a resource manager to manage the resources specified by the array *rsrc.* The number of resources (ie. the length of the array) is specified by *cnt.* The *desc* is an optional string used for naming the latches protecting the resources. It can be useful in debugging.

~rsrc_m()

The destructor destroys the resource manager. There should not be any resources pinned when the resource manager is is destroyed.

grab(ret, key, found, is_new, mode, timeout)

The **grab** method pins the resource associated with *key* and sets a latch in mode *mode* on the resource. The calling thread should subsequently free *rsrc* by calling **unpin.**

If the resource is cached, **grab** simply returns it. Otherwise, **grab** will either allocate an unused resource or find another cached resource to replace using a pseudo-LRU (clock) algorithm. The calling thread could potentially block if *mode* causes a latch conflict (i.e., when there is contention to the resource). If **grab** is successful, a pointer to the cached/allocated/replacement resource is returned in *ret.* The *found* flag is set to indicate cache hit/miss. In the case of a cache miss, the resource returned is said to be **in-transit,** and the *is_new* flag indicates whether *ret* points to:

(1) a previously unused resource (true), or

(2) a previously cached resource of another key (false).

In case 1, the in-transit resource returned simply needs to be initialized with the new key. All other threads that ask for a resource with the new key will block. The caller should initialize the resource and subsequently call **publish,** which formally publishes the new key and resets the resource's in-transit status.

In case 2, the in-transit resource returned is temporarily associated with both the new key (as specified in **grab** and the old key. All other threads that ask for a resource with any of these keys will block. The caller sehould first clean up the resource (invalidate the old key) and call **publish_partial,** which informs **rsrc_m** that the old key is no longer valid. The caller should then proceed as in case 1.

In essense, the caller should proceed as follows:

```
grab the resource
if not found then
  if not is_new then
      clean up the resource (optional), e.g.,flush the dirty page
      call publish_partial() (optional)
  initialize the resource (obligatory), e.g.,read the new page
  call publish() (obligatory)
... use the resource ...
call unpin() to free the resource
```

find(ret, key, mode, ref_bit, timeout)

The **find** method looks up and pins a cached resource identified by *key.* It returns an the error **fcNOTFOUND**
if the resource is not cached. If the resource is cached, a *mode*
latch is acquired on the resource and a pointer to the resource is returned in *ret.* The calling thread should subsequently free the resource by calling **unpin.** As in **grab,**
the calling thread could potentially block if *mode* causes a latch conflict (i.e.,when there is contention to the resource). The *refbit* parameter is a hint to the **rsrc_m** replacement algorithm; *refbit* is directly proportional to the duration that a resource remained cached. Thus, a zero *refbit* implies that the **rsrc_m** should reuse the resource as soon as needed after it is unpinned.

pin(rsrc, mode)

The **pin** method pins the resource *rsrc.* The latch on the resource is acquired in mode *mode.* The calling thread should subsequently free *rsrc* by calling **unpin.**

publish(rsrc, error_flag)

The **publish** method makes the resource *rsrc,* that was previously obtained by a **grab** call with a cache miss, available. See the description of **grab** for more details. The *error_flag* parameter is informs the **rsrc_m** that the resource has not been successfully initialized, and should be invalidated.

publish_partial(rsrc)

The **publish_partial** method partially publishes the resource *rsrc* that was previously obtained with a call to **grab.** See the description of **grab** for more details.

unpin(rsrc, refbit)

The **unpin** method releases the latch on the resource *rsrc.* The *refbit* parameter is a hint to the **rsrc_m** replacement algorithm; *refbit* is directly proportional to the duration that a resource remained cached. Thus, a zero *refbit* implies that the **rsrc_m** should reuse the resource as soon as needed.

Rsrc_i Interface

The **rsrc_i** template is used to iterate over all of the resources in an instance of **rsrc_m.**

rsrc_m(r, mode, start)

The constructor initilizes an iterator for the **rsrc_m** instance indicated by parameter *r.* Each resource will be pinned (latched) in mode *mode.* The iterator starts at the *start,* element in the array of resources that *r* manages. The iterator will only return those resources actually in the hash table.

˜rsrc_m()

The destructor ends the iterator by unpinning and currently pinned resource.

next()

The **next** method unpins the current resource, advances the iterator to the next resource, and pins it. **Next** returns a pointer to the resource after it has advanced. It will return 0 if there are no more resources. **Next** skips any resources not in the hash table.

curr()

The **curr** method returns a pointer to the currently pinned resource.

discard_curr()

The **discard_curr** method unpins the current resource and removes it from the hash table.

TODO

VERSION

This manual page applies to Version 1.1 of the Shore software.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

COPYRIGHT

SEE ALSO

latch_t(common), intro(common).