Trove DBPF Handle Allocator

PVFS Development Team

March 2, 2009

$Id: handle-allocator.tex,v 1.1 2003/01/24 23:29:18 pcarns Exp $

1 Introduction

The Trove interface gives out handles – unique identifiers to trove objects. In addition to being unique, handles will not be reused within a configurable amount of time. These two constraints make for a handle allocator that ends up being a bit more complicated than one might expect. Add to that the fact that we want to serialize on disk all or part of the handle allocator's state, and here we are with a document to explain it all.

1.1 Data Structures

1.1.1 Extents

We have a large handle space we need to represent efficiently. This approach uses extents:

```
struct extent {
int64_t first;
int64_t last;
};
```

1.1.2 Extent List

We keep the extents (not nescessarily sorted) in the `extents` array. For faster searches, `index` keeps an index into `extents` in an AVL tree. In addition to the extents themselves, some bookkeeping members are added. The most important is the `timestamp` member, used to make sure no handle in its list gets reused before it should. `__size` is only used internally, keeping track of how big `extents` is.

```
struct extentlist {
int64_t __size;
int64_t num_extents;
```

```
int64_t num_handles;
struct timeval timestamp;
struct extent * extents;
};
```

1.1.3 Handle Ledger

We manage several lists. The `free_list` contains all the valid handles. The `recently_freed_list` contains handles which have been freed, but possibly before some expire time has passed. The `overflow_list` holds freed handles while items on the `recently_freed_list` wait for the expire time to pass.

We save our state by writing out and reading from the three `TROVE_handle` members, making use of the higher level trove interface.

```
struct handle_ledger {
        struct extentlist free_list;
struct extentlist recently_freed_list;
struct extentlist overflow_list;
FILE *backing_store;
TROVE_handle free_list_handle;
TROVE_handle recently_freed_list_handle;
TROVE_handle overflow_list_handle;
}
```

2 Algorithm

2.1 Assigning handles

Start off with a `free_list` of one big extent encompassing the entire handle space.

- Get the last extent from the `free_list` (We hope getting the last extent improves the effiency of the extent representation)
- Save `last` for later return to the caller
- Decrement `last`
- if $first > last$, mark the extent as empty.

2.2 returning handles

- when the first handle is returned, it gets added to the `recently_freed` list. Because this is the first item on that list, we check the time.
- now we add more handles to the list. we check the time after N handles are returned and update the timestamp.

- Once we have added H handles, we decide the `recently_freed` list has enough handles. We then start using the `overflow_list` to hold returned handles.
- as with the `recently_freed` list, we record the time that this handle was added, updating the timestamp after every N additions. We also check how old the `recently_freed` list is.
- at some point in time, the whole `recently_freed` list is ready to be returned to the `free_list`. The `recently_freed` list is merged into the `free_list`, the `overflow_list` becomes the `recently_freed` list and the `overflow_list` is empty.

2.3 I don't know what to call this section

Let T_r be the minimum response time for an operation of any sort, T_f be the time a handle must sit before being moved back to the free list, and N_{tot} be the total number of handles available on a server.

The pathological case would be one where a caller

- fills up the `recently_freed` list
- immediately starts consuming handles as quickly as possible to make for the largest possible `recently_freed` list in the next pass

This results in the largest number of handles being unavailable due to sitting on the `overflow_list`. Call N_{purg} the number of handles waiting in "purgatory" (waiting for T_f to pass)

$$N_{purg} = T_f / T_r \tag{1}$$

$$F_{purg} = N_{purg} / N_{tot} \tag{2}$$

$$F_{purg} = T_f / (T_r * N_{tot}) \tag{3}$$

We should try to collect statistics and see what T_r and N_{purg} end up being for real and pathological workloads.