



INTEL® PMDK WORKSHOP

英特尔®PMDK 研讨会

LIBMEMOBJ-CPP API

<https://github.com/pmem/libpmemobj-cpp>

Speaker : Igor Chorażewicz
igor.chorazewicz@intel.com

List of content

- pool<> class
- transactions
- p<> property
- peristent pointer
- transactional allocation
- persistent memory synchronization
- persistent container - array
- persistent container - vector

pmem::obj::pool

- Class template, where the template parameter is the type of the root object
- Supports three basic operations
 - open – opens an existing pmempobj pool
 - create – creates a new pmemobj pool
 - close – closes an already opened/created pool
- Inherits from pool_base

pmem::obj::pool example

```
if (access(path.c_str(), F_OK) != 0) {  
    pop = pool<root>::create(path, "some_layout", PMEMOBJ_MIN_POOL,  
                               S_IRWXU);  
} else {  
    pop = pool<root>::open(path, "some_layout");  
}  
  
// get root object  
auto r = pop.root();
```

Transactions

Introduction to transactions

- Undo log based transactions
 - In case of interruption it is rolled-back or completed upon next pool open
- ACID like properties
- Can be nested
- Locks are held until the end of a transaction

Closure transactions example

```
auto pop = pool<root>::open("/path/to/poolfile", "layout string");  
transaction::run(pop, [] {  
    // do some work...  
}, persistent_mtx, persistent_shmtx);
```


Closure transactions

- Take an `std::function` object as transaction body
- No explicit transaction commit
- Available with every C++11 compliant compiler
- Throw an exception when the transaction is aborted
- Take an arbitrary number of locks
 - Unfortunately at the very end

Manual transaction example

```
auto pop = pool::open("/path/to/poolfile", "layout string");  
  
{  
    transaction::manual(pop, persistent_mtx, persistent_shmtx);  
    // do some work...  
    transaction::commit();  
}  
  
auto aborted = transaction::error();
```

Manual transactions

- Based on the familiar RAII concept
- Fairly easy to use
- Explicit transaction commit because of `std::uncaught_exception`
- Does not throw an exception on transaction abort
- By default aborts to account for third-party exceptions or amnesia
- Accepts an arbitrary number of (persistent memory resident) locks

Automatic transactions example

```
auto pop = pool<root>::open("/path/to/poolfile", "layout string");  
try {  
    transaction::automatic(pop, persistent_mtx, persistent_shmtx);  
    // do some work...  
} catch (...) {  
    // do something meaningful  
}
```


Automatic transactions

- Functionally and semantically almost identical to the manual transaction
- No explicit transaction commit
- Need C++17
- Relies on `std::uncaught_exceptions`

pmem::obj::p<>

pmem::obj::p

- AKA the workhorse
- Overloads operator= for snapshotting in a transaction
- Overloads a bunch of other operators for seamless integration
 - Arithmetic
 - Logical
- Should be used for fundamental types
- No convenient way to access members of aggregate types
- No operator. to overload

Code with manual snapshotting

```
struct data {  
    int x;  
}  
  
auto pop = pool<data>::("/path/to/poolfile", "layout string");  
auto datap = pop.root();  
  
transaction::run(pop, [&]{  
    pmemobj_tx_add_range(root, 0, sizeof (struct data));  
    datap->x = 5;  
});
```


Code with pmem::obj:p

```
struct data {  
    p<int> x;  
}  
  
auto pop = pool<data>::("/path/to/poolfile", "layout string");  
auto datap = pop.root();  
  
transaction::run(pop, [&]{  
    datap->x = 5;  
});
```

Persistent pointer

pmem::obj::persistent_ptr

- Points to objects within a persistent memory pool
 - Manages translating persistent addresses to runtime addresses
- Is a random access iterator
- Has primitives for flushing contents to persistence

pmem::obj::persistent_ptr

- Does not manage object lifetime
- Does not automatically add contents to the transaction
 - But it does add itself to the transaction
- Does not point to polymorphic objects
 - No good way to rebuild runtime state after pool reboot

Transactional allocation

- Can be used only within transactions
- Use transaction logic to enable allocation/delete rollback of persistent state
- `make_persistent` calls appropriate constructor
 - Syntax similar to `std::make_shared`
- `delete_persistent` calls the destructor
 - Not similar to anything found in `std`

Transactional allocation example

```
struct data {  
    data(int a, int b) : a(a), b(b) {}  
    int a;  
    int b;  
}  
  
transaction::run(pop, [&]{  
    persistent_ptr<data> ptr = make_persistent<data>(1, 2);  
    assert(ptr->a == 1);  
    assert(ptr->b == 2);  
  
    persistent_ptr<data> ptr2 = make_persistent<data>(allocation_flag::no_flush(),  
                                                       2, 3);  
  
    ...  
  
    delete_persistent<data>(ptr);  
});
```

Allocation flags

- `class_id(id)`
 - Allocate the object from the allocation class with id equal to id
- `no_flush()`
 - Skip flush on commit

Thread synchronization

Persistent Memory Synchronization

- Types:
 - mutex
 - shared_mutex
 - timed_mutex
 - condition_variable
- All with an interface similar to their std counterparts
- Auto reinitializing
- Can be used with transactions

Persistent memory containers

pmem::obj::experimental::array

- std::array compatible interface (almost)
- Takes care of adding elements to a transaction
 - In operator[]/at() when obtaining non-const reference
 - On iterator dereference
 - In other methods which allow write access to data
- Works with std algorithms

pmem::obj::experimental::array example

```
transaction::run(pop, [&]{  
    auto ptr = make_persistent<array<int, 6>>();  
  
    // iterators will snapshot on element access  
    std::fill(ptr->begin(), ptr->end(), 1);  
  
    // modify all elements in a range  
    for (auto &e : ptr->range(0, 3)) {  
        e++;  
    }  
  
    delete_persistent<array<int, 6>>(ptr);  
});
```


pmem::obj::experimental::vector

- `std::vector` compatible interface (almost)
- Takes care of adding elements to a transaction
 - The same way as in array
- All functions which may alter vector properties are atomic
 - This includes: `resize()`, `reserve()`, `push_back()` and others
 - Transactions are used internally
 - Strong exception gurantee

pmem::obj::experimental::vector example

```
transaction::run(pop, [&]{  
    auto ptr = make_persistent<vector<int>>();  
  
    ptr->push_back(1);  
  
    ptr->resize(10);  
    ptr->at(5) = 10;  
  
    delete_persistent<vector<int>>(ptr);  
});
```

Types requirements

Types requirements for peristent objects

- Maximum size equals to PMEMOBJ_MAX_ALLOC_SIZE
- Should satisfy StandardLayoutType requirement
 - Object representation might differ between compilers
- Should satisfy TriviallyCopyable
 - Library does not call constructors/destructors during snapshotting

Q&A





谢谢