CHAPTER 14

Concurrency and Persistent Memory

This chapter discusses what you need to know when building multithreaded applications for persistent memory. We assume you already have experience with multithreaded programming and are familiar with basic concepts such as mutexes, critical section, deadlocks, atomic operations, and so on.

The first section of this chapter highlights common practical solutions for building multithreaded applications for persistent memory. We describe the limitation of the Persistent Memory Development Kit (PMDK) transactional libraries, such as `libpmemobj` and `libpmemobj-cpp`, for concurrent execution. We demonstrate simple examples that are correct for volatile memory but cause data inconsistency issues on persistent memory in situations where the transaction aborts or the process crashes. We also discuss why regular mutexes cannot be placed as is on persistent memory and introduce the persistent deadlock term. Finally, we describe the challenges of building lock-free algorithms for persistent memory and continue our discussion of visibility vs. persistency from previous chapters.

The second section demonstrates our approach to designing concurrent data structures for persistent memory. At the time of publication, we have two concurrent associative C++ data structures developed for persistent memory - a concurrent hash map and a concurrent map. More will be added over time. We discuss both implementations within this chapter.

All code samples are implemented in C++ using the `libpmemobj-cpp` library described in Chapter 8. In this chapter, we usually refer to `libpmemobj` because it implements the features and `libpmemobj-cpp` is only a C++ extension wrapper for it. The concepts are general and can apply to any programming language.

S. Scargall, *Programming Persistent Memory*, https://doi.org/10.1007/978-1-4842-4932-1_14

Transactions and Multithreading

In computer science, ACID (atomicity, consistency, isolation, and durability) is a set of properties of transactions intended to guarantee data validity and consistency in case of errors, power failures, and abnormal termination of a process. Chapter 7 introduced PMDK transactions and their ACID properties. This chapter focuses on the relevancy of multithreaded programs for persistent memory. Looking forward, Chapter 16 will provide some insights into the internals of `libpmemobj` transactions.

The small program in Listing 14-1 shows that the `counter` stored within the `root` object is incremented concurrently by multiple threads. The program opens the persistent memory pool and prints the value of `counter`. It then runs ten threads, each of which calls the `increment()` function. Once all the threads complete successfully, the program prints the final value of `counter`.

Listing 14-1. Example to demonstrate that PMDK transactions do not automatically support isolation

```
41  using namespace std;
42  namespace pobj = pmem::obj;
43
44  struct root {
45      pobj::p<int> counter;
46  };
47
48  using pop_type = pobj::pool<root>;
49
50  void increment(pop_type &pop) {
51      auto proot = pop.root();
52      pobj::transaction::run(pop, [&] {
53          proot->counter.get_rw() += 1;
54      });
55  }
56
57  int main(int argc, char *argv[]) {
58      pop_type pop =
59          pop_type::open("/pmemfs/file", "COUNTER_INC");
60
```

```
61      auto proot = pop.root();
62
63      cout << "Counter = " << proot->counter << endl;
64
65      std::vector<std::thread> workers;
66      workers.reserve(10);
67      for (int i = 0; i < 10; ++i) {
68          workers.emplace_back(increment, std::ref(pop));
69      }
70
71      for (int i = 0; i < 10; ++i) {
72          workers[i].join();
73      }
74
75      cout << "Counter = " << proot->counter << endl;
76
77      pop.close();
78      return 0;
79  }
```

You might expect that the program in Listing 14-1 the prints a final counter value of 10. However, PMDK transactions do not automatically support isolation from the ACID properties set. The result of the increment operation on line 53 is visible to other concurrent transactions before the current transaction has implicitly committed its update on line 54. That is, a simple data race is occurring in this example. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the operating system's thread scheduling algorithm can swap between threads at any time, there is no way for the application to know the order in which the threads will attempt to access the shared data. Therefore, the result of the change of the data is dependent on the thread scheduling algorithm, that is, both threads are "racing" to access/change the data.

If we run this example multiple times, the results will vary from run to run. We can try to fix the race condition by acquiring a mutex lock before the counter increment as shown in Listing 14-2.

Listing 14-2. Example of incorrect synchronization inside a PMDK transaction

```
46  struct root {
47      pobj::mutex mtx;
48      pobj::p<int> counter;
49  };
50
51  using pop_type = pobj::pool<root>;
52
53  void increment(pop_type &pop) {
54      auto proot = pop.root();
55      pobj::transaction::run(pop, [&] {
56          std::unique_lock<pobj::mutex> lock(proot->mtx);
57          proot->counter.get_rw() += 1;
58      });
59  }
```

- Line 47: We added a `mutex` to the root data structure.
- Line 56: We acquired the mutex lock within the transaction before incrementing the value of `counter` to avoid a race condition. Each thread increments the counter inside the critical section protected by the mutex.

Now if we run this example multiple times, it will always increment the value of the `counter` stored in persistent memory by 1. But we are not done yet. Unfortunately, the example in Listing 14-2 is also wrong and can cause data inconsistency issues on persistent memory. The example works well if there are no transaction aborts. However, if the transaction aborts after the lock is released but before the transaction has completed and successfully committed its update to persistent memory, other threads can read a cached value of the counter that can cause data inconsistency issues. To understand the problem, you need to know how `libpmemobj` transactions work internally. For now, we discuss only the necessary details required to understand this issue and leave the in-depth discussion of transactions and their implementation for Chapter 16.

A `libpmemobj` transaction guarantees atomicity by tracking changes in the undo log. In the case of a failure or transaction abort, the old values for uncommitted changes are restored from the undo log. It is important to know that the undo log is a thread-specific

entity. This means that each thread has its own undo log that is not synchronized with undo logs of other threads.

Figure 14-1 illustrates the internals of what happens within the transaction when we call the `increment()` function in Listing 14-2. For illustrative purposes, we only describe two threads. Each thread executes concurrent transactions to increment the value of `counter` allocated in persistent memory. We assume the initial value of `counter` is `0` and the first thread acquires the lock, while the second thread waits on the lock. Inside the critical section, the first thread adds the initial value of `counter` to the undo log and increments it. The mutex is released when execution flow leaves the lambda scope, but the transaction has not committed the update to persistent memory. The changes become immediately visible to the second thread. After a user-provided lambda is executed, the transaction needs to flush all changes to persistent memory to mark the change(s) as committed. Concurrently, the second thread adds the current value of `counter`, which is now 1, to its undo log and performs the increment operation. At that moment, there are two uncommitted transactions. The undo log of Thread 1 contains `counter = 0`, and the undo log of Thread 2 contains `counter = 1`. If Thread 2 commits its transaction while Thread 1 aborts its transaction for some reason (crash or abort), the incorrect value of `counter` will be restored from the undo log of Thread 1.

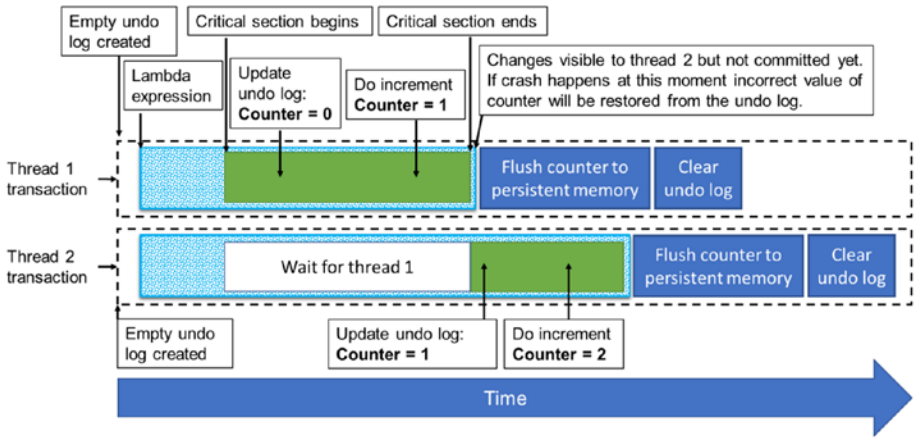

Figure 14-1. *Illustrative execution of the Listing 14-2 example*

The solution is to hold the mutex until the transaction is fully committed, and the data has been successfully flushed to persistent memory. Otherwise, changes made by one transaction become visible to concurrent transactions before it is persisted and committed. Listing 14-3 demonstrates how to implement the `increment()` function correctly.

Listing 14-3. Correct example for concurrent PMDK transaction

```
52  void increment(pop_type &pop) {
53      auto proot = pop.root();
54      pobj::transaction::run(pop, [&] {
55          proot->counter.get_rw() += 1;
56      }, proot->mtx);
57  }
```

The `libpmemobj` API allows us to specify locks that should be acquired and held for the entire duration of the transaction. In the Listing 14-3 example, we pass the `proot->mtx` mutex object to the `run()` method as a third parameter.

Mutexes on Persistent Memory

Our previous examples used `pmem::obj::mutex` as a type for the `mtx` member in our `root` data structure instead of the regular `std::mutex` provided by Standard Template Library. The `mtx` object is a member of the `root` object that resides in persistent memory. The `std::mutex` type cannot be used on persistent memory because it may cause persistent deadlock.

A persistent deadlock happens if an application crash occurs while holding a mutex. When the program starts, if it does not release or reinitialize the mutex at startup, threads that try to acquire it will wait forever. To avoid such situations, `libpmemobj` provides synchronization primitives that reside in persistent memory. The main feature of synchronization primitives is that they are automatically reinitialized every time the persistent object store pool is open.

For C++ developers, the `libpmemobj-cpp` library provides C++11-like synchronization primitives shown in Table 14-1.

Table 14-1. *Synchronization primitives provided by libpmemob++ library*

Class	Description
pmem::obj::mutex	This class is an implementation of a persistent memory resident mutex which mimics in behavior the C++11 std::mutex. This class satisfies all requirements of the Mutex and StandardLayoutType concepts.
pmem::obj::timed_mutex	This class is an implementation of a persistent memory resident timed_mutex which mimics in behavior the C++11 std::timed_ mutex. This class satisfies all requirements of TimedMutex and StandardLayoutType concepts.
pmem::obj::shared_mutex	This class is an implementation of a persistent memory resident shared_mutex which mimics in behavior the C++17 std::shared_ mutex. This class satisfies all requirements of SharedMutex and StandardLayoutType concepts.
pmem::obj:: condition_variable	This class is an implementation of a persistent memory resident condition variable which mimics in behavior the C++11 std::condition_variable. This class satisfies all requirements of StandardLayoutType concept.

For C developers, the `libpmemobj` library provides pthread-like synchronization primitives shown in Table 14-2. Persistent memory-aware locking implementations are based on the standard POSIX Thread Library and provide semantics similar to standard pthread locks.

Table 14-2. *Synchronization primitives provided by the libpmemobj library*

Structure	Description
PMEMmutex	The data structure represents a persistent memory resident mutex similar to `pthread_mutex_t`.
PMEMrwlock	The data structure represents a persistent memory resident read-write lock similar to `pthread_rwlock_t`.
PMEMcond	The data structure represents a persistent memory resident condition variable similar to `pthread_cond_t`.

These convenient persistent memory-aware synchronization primitives are available for C and C++ developers. But what if a developer wants to use a custom synchronization object that is more appropriate for a particular use case? As we mentioned earlier, the main feature of persistent memory-aware synchronization primitives is that they are reinitialized every time we open a persistent memory pool. The `libpmemobj-cpp` library provides a more generic mechanism to reinitialize any user-provided type every time a persistent memory pool is opened.

The `libpmemobj-cpp` provides the `pmem::obj::v<T>` class template which allows creating a volatile field inside a persistent data structure. The mutex object is semantically a volatile entity, and the state of a mutex should not survive an application restart. On application restart, a mutex object should be in the unlocked state. The `pmem::obj::v<T>` class template is targeted for this purpose. Listing 14-4 demonstrates how to use the `pmem::obj::v<T>` class template with `std::mutex` on persistent memory.

Listing 14-4. Example demonstrating usage of std::mutex on persistent memory

```
38  namespace pobj = pmem::obj;
39
40  struct root {
41      pobj::experimental::v<std::mutex> mtx;
42  };
43
44  using pop_type = pobj::pool<root>;
45
46  int main(int argc, char *argv[]) {
47      pop_type pop =
48          pop_type::open("/pmemfs/file", "MUTEX");
49
50      auto proot = pop.root();
51
52      proot->mtx.get().lock();
53
54      pop.close();
55      return 0;
56  }
```

- Line 41: We are only storing the `mtx` object inside root object on persistent memory.
- Lines 47-48: We open the persistent memory pool with the layout name of "MUTEX".
- Line 50: We obtain a pointer to the root data structure within the pool.
- Line 52: We acquire the mutex.
- Lines 54-56: Close the pool and exit the program.

As you can see, we do not explicitly unlock the mutex within the `main()` function. If we run this example several times, the `main()` function can always lock the mutex on line 52. This works because the `pmem::obj::v<T>` class template implicitly calls a default constructor, which is a wrapped `std::mutex` object type. The constructor is called every time we open the persistent memory pool so we never run into a situation where the lock is already acquired.

If we change the `mtx` object type on line 41 from `pobj::experimental::v<std::mutex>` to `std::mutex` and try to run the program again, the example will hang during the second run on line 52 because `mtx` object was locked during the first run and we never released it.

Atomic Operations and Persistent Memory

Atomic operations cannot be used inside PMDK transactions for the reason described in Figure 14-1. Changes made by atomic operations inside a transaction become visible to other concurrent threads before the transaction is committed. It forces data inconsistency issues in cases of abnormal program termination or transaction aborts. Consider lock-free algorithms where concurrency is achieved by atomically updating the state in memory.

Lock-Free Algorithms and Persistent Memory

It is intuitive to think that lock-free algorithms are naturally fit for persistent memory. In lock-free algorithms, thread-safety is achieved by atomic transitions between consistent states, and this is exactly what we need to support data consistency in persistent memory. But this assumption is not always correct.

To understand the problem with lock-free algorithms, remember that a system with persistent memory will usually have the virtual memory subsystem divided into two domains: volatile and persistent (described in Chapter 2). The result of an atomic operation may only update data in a CPU cache using a cache coherency protocol. There is no guarantee that the data will be flushed unless an explicit flush operation is called. CPU caches are only included within the persistence domain on platforms with eADR support. This is not mandatory for persistent memory. ADR is the minimal platform requirement for persistent memory, and in that case, CPU caches are not flushed in a power failure.

Figure 14-2 assumes a system with ADR support. The example shows concurrent lock-free insert operations to a singly linked list located in persistent memory. Two threads are trying to insert new nodes to the tail of a linked list using a compare-and-exchange (`CMPXCHG` instruction) operation followed by a cache flush operation (`CLWB` instruction). Assume Thread 1 succeeds with its compare-and-exchange, so the change appears in a volatile domain and becomes visible to the second thread. At this moment, Thread 1 may be preempted (changes not flushed to a persistent domain), while Thread 2 inserts Node 5 after Node 4 and flushes it to a persistent domain. A possibility for data inconsistency exists because Thread 2 performed an update based on the data that is not yet persisted by Thread 1.

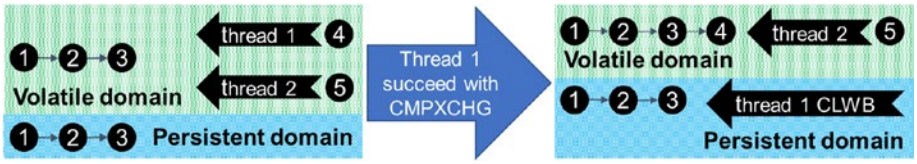

Figure 14-2. *Example of a concurrent lock-free insert operation to a singly linked list located in persistent memory*

Concurrent Data Structures for Persistent Memory

This section describes two concurrent data structures available in the `libpmemobj-cpp` library: `pmem::obj::concurrent_map` and `pmem::obj::concurrent_hash_map`. Both are associative data structures composed of a collection of key and value pairs, such that each possible key appears at most once in the collection. The main difference between them is that the concurrent hash map is unordered, while the concurrent map is ordered by keys.

We define *concurrent* in this context to be the method of organizing data structures for access by multiple threads. Such data structures are intended for use in a parallel computing environment when multiple threads can concurrently call methods of a data structure without additional synchronization required.

C++ Standard Template Library (STL) data structures can be wrapped in a coarse-grained mutex to make them safe for concurrent access by letting only one thread operate on the container at a time. However, that approach eliminates concurrency and thereby restricts parallel speedup if implemented in performance-critical code. Designing concurrent data structures is a challenging task. The difficulty increases significantly when we need to develop concurrent data structures for persistent memory and make them fault tolerant.

The `pmem::obj::concurrent_map` and `pmem::obj::concurrent_hash_map` structures were inspired by the Intel Threading Building Blocks (Intel TBB),[1] which provides implementations of these concurrent data structures designed for volatile memory. You can read the *Pro TBB: C++ Parallel Programming with Threading Building* Blocks book[2] to get more information and learn how to use these concurrent data structures in your application. The free electronic copy is available from Apress at https://www.apress.com/gp/book/9781484243978.

There are three main methods in our concurrent associative data structures: `find`, `insert`, and `erase/delete`. We describe each data structure with a focus on these three methods.

Concurrent Ordered Map

The implementation of the concurrent ordered map for persistent memory (`pmem::obj::concurrent_map`) is based on a concurrent skip list data structure. Intel TBB supplies `tbb::concurrent_map`, which is designed for volatile memory that we use as a baseline for a port to persistent memory. The concurrent skip list data structure can be implemented as a lock-free algorithm. But Intel chose a provably correct

[1]Intel Threading Building Blocks library (https://github.com/intel/tbb).

[2]Michael Voss, Rafael Asenjo, James Reinders. C++ Parallel Programming with Threading Building Blocks; Apress, 2019; ISBN-13 (electronic): 978-1-4842-4398-5; https://www.apress.com/gp/book/9781484243978.

scalable concurrent skip list[3] implementation with fine-grain locking distinguished by a combination of simplicity and scalability. Figure 14-3 demonstrates the basic idea of the skip list data structure. It is a multilayered linked list-like data structure where the bottom layer is an ordered linked list. Each higher layer acts as an "express lane" for the following lists and allows it to skip elements during lookup operations. An element in layer `i` appears in layer `i+1` with some fixed probability `p` (in our implementation `p = 1/2`). That is, the frequency of nodes of a particular height decreases exponentially with the height. Such properties allow it to achieve O(log n) average time complexity for lookup, insert, and delete operations. O(log n) means the running time grows at most proportional to *"log n"*. You can learn more about *Big O notation* on Wikipedia at `https://en.wikipedia.org/wiki/Big_O_notation`

For the implementation of `pmem::obj::concurrent_map`, the `find` and `insert` operations are thread-safe and can be called concurrently with other `find` and `insert` operations without requiring additional synchronizations.

Find Operation

Because the `find` operation is non-modifying, it does not have to deal with data consistency issues. The `lookup` operation for the target element always begins from the topmost layer. The algorithm proceeds horizontally until the next element is greater or equal to the target. Then it drops down vertically to the next lower list if it cannot proceed on the current level. Figure 14-3 illustrates how the `find` operation works for the element with key=9. The search starts from the highest level and immediately goes from dummy head node to the node with key=4, skipping nodes with keys 1, 2, 3. On the node with key=4, the search is dropped two layers down and goes to the node with key=8. Then it drops one more layer down and proceeds to the desired node with key=9.

[3]M. Herlihy, Y. Lev, V. Luchangco, N. Shavit. A provably correct scalable concurrent skip list. In OPODIS '06: Proceedings of the 10th International Conference On Principles Of Distributed Systems, 2006; `https://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/OPODIS2006-BA.pdf`.

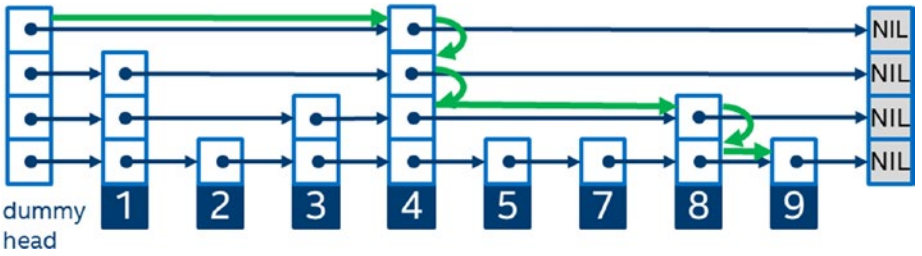

Figure 14-3. *Finding key=9 in the skip list data structure*

The `find` operation is wait-free. That is, every `find` operation is bound only by the number of steps the algorithm takes. And a thread is guaranteed to complete the operation regardless of the activity of other threads. The implementation of `pmem::obj::concurrent_map` uses atomic load-with-acquire memory semantics when reading pointers to the next node.

Insert Operation

The `insert` operation, shown in Figure 14-4, employs fine-grained locking schema for thread-safety and consists of the following basic steps to insert a new node with key=7 into the list:

1. Allocate the new node with randomly generated height.
2. Find a position to insert the new node. We must find the predecessor and successor nodes on each level.
3. Acquire locks for each predecessor node and check that the successor nodes have not been changed. If successor nodes have changed, the algorithm returns to step 2.
4. Insert the new node to all layers starting from the bottom one. Since the *find* operation is lock-free, we must update pointers on each level atomically using store-with-release memory semantics.

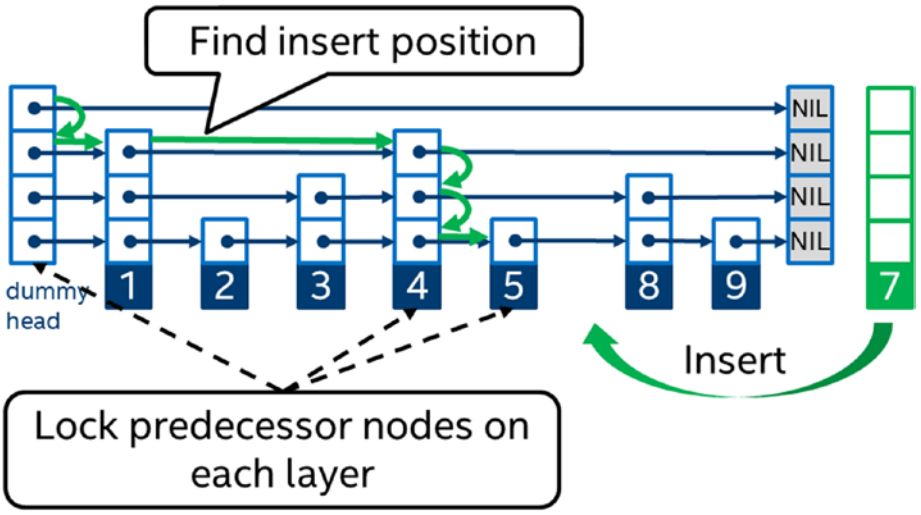

Figure 14-4. *Inserting a new node with key=7 into the concurrent skip list*

The algorithm described earlier is thread-safe, but it is not enough to be fault tolerant on persistent memory. There is a possible persistent memory leak if a program unexpectedly terminates between the first and fourth steps of our algorithm.

The implementation of `pmem::obj::concurrent_map` does not use transactions to support data consistency because transactions do not support isolation and by not using transactions, it can achieve better performance. For this linked list data structure, data consistency is maintained because a newly allocated node is always reachable (to avoid persistent memory leak) and the linked list data structure is always valid. To support these two properties, persistent thread-local storage is used, which is a member of the concurrent skip list data structure. Persistent thread-local storage guarantees that each thread has its own location in persistent memory to assign the result of persistent memory allocation for the new node.

Figure 14-5 illustrates the approach of this fault-tolerant `insert` algorithm. When a thread allocates a new node, the pointer to that node is kept in persistent thread-local storage, and the node is reachable through this persistent thread-local storage. Then the algorithm inserts the new node to the skip list by linking it to all layers using the thread-safe algorithm described earlier. Finally, the pointer in the persistent thread-local storage is removed because the new node is reachable now via skip list itself. In case of failure, a special function traverses all nonzero pointers in persistent thread-local storage and completes the `insert` operation.

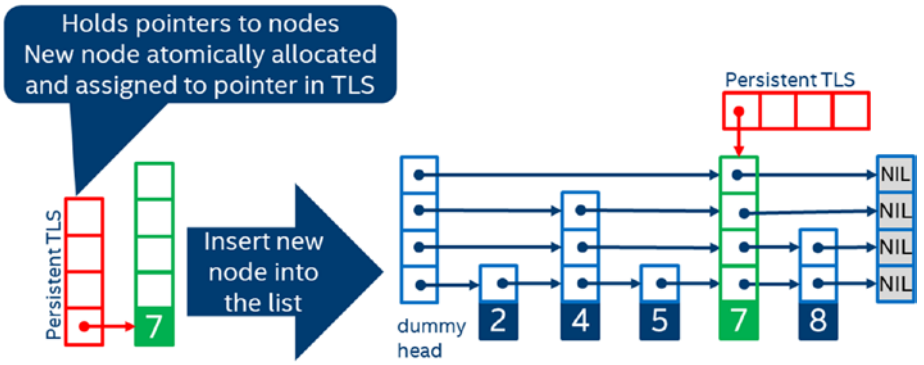

Figure 14-5. *Fault-tolerant insert operation using persistent thread-local storage*

Erase Operation

The implementation of the `erase` operation for `pmem::obj::concurrent_map` is not thread-safe. This method cannot be called concurrently with other methods of the concurrent ordered map because this is a memory reclamation problem that is hard to solve in C++ without a garbage collector. There is a way to logically extract a node from a skip list in a thread-safe manner, but it is not trivial to detect when it is safe to delete the removed node because other threads may still have access to the node. There are possible solutions, such as hazard pointers, but these can impact the performance of the `find` and `insert` operations.

Concurrent Hash Map

The concurrent hash map designed for persistent memory is based on `tbb::concurrent_hash_map` that exists in the Intel TBB. The implementation is based on a concurrent hash table algorithm where elements assigned to buckets based on a hash code are calculated from a key. In addition to concurrent `find, insert`, and `erase` operations, the algorithm employs concurrent resizing and on-demand per-bucket rehashing.[4]

Figure 14-6 illustrates the basic idea of the concurrent hash table. The hash table consists of an array of buckets, and each bucket consists of a list of nodes and a read-write lock to control concurrent access by multiple threads.

[4]Anton Malakhov. Per-bucket concurrent rehashing algorithms, 2015, arXiv:1509.02235v1; https://arxiv.org/ftp/arxiv/papers/1509/1509.02235.pdf.

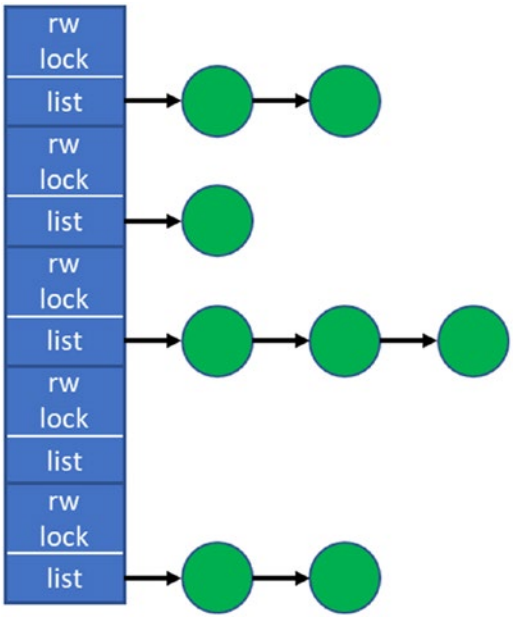

Figure 14-6. *The concurrent hash map data structure*

Find Operation

The `find` operation is a read-only event that does not change the hash map state. Therefore, data consistency is maintained while performing a `find` request. The `find` operation works by first calculating the hash value for a target key and acquires read lock for the corresponding bucket. The read lock guarantees that there is no concurrent modifications to the bucket while we are reading it. Inside the bucket, the `find` operation performs a linear search through the list of nodes.

Insert Operation

The `insert` method of the concurrent hash map uses the same technique to support data consistency as the concurrent skip list data structure. The operation consists of the following steps:

1. Allocate the new node, and assign a pointer to the new node to persistent thread-local storage.
2. Calculate the hash value of the new node, and find the corresponding bucket.

3. Acquire the write lock to the bucket.
4. Insert the new node to the bucket by linking it to the list of nodes. Because only one pointer has to be updated, a transaction is not needed. Because only one pointer is updated, a transaction is not required.

Erase Operation

Although the `erase` operation is similar to an `insert` (the opposite action), its implementation is even simpler than the `insert`. The `erase` implementation acquires the write lock for the required bucket and, using a transaction, removes the corresponding node from the list of nodes within that bucket.

Summary

Although building an application for persistent memory is a challenging task, it is more difficult when you need to create a multithreaded application for persistent memory. You need to handle data consistency in a multithreaded environment when multiple threads can update the same data in persistent memory.

If you develop concurrent applications, we encourage you to use existing libraries that provide concurrent data structures designed to store data in persistent memory. You should develop custom algorithms only if the generic ones do not fit your needs. See the implementations of concurrent `cmap` and `csmap` engines in `pmemkv`, described in Chapter 9, which are implemented using `pmem::obj::concurrent_hash_map` and `pmem::obj::concurrent_map`, respectively.

If you need to develop a custom multithreaded algorithm, be aware of the limitation PMDK transactions have for concurrent execution. This chapter shows that transactions do not automatically provide isolation out of the box. Changes made inside one transaction become visible to other concurrent transactions before they are committed. You will need to implement additional synchronization if it is required by an algorithm. We also explain that atomic operations cannot be used inside a transaction while building lock-free algorithms without transactions. This is a very complicated task if your platform does not support eADR.