CHAPTER 16

PMDK Internals: Important Algorithms and Data Structures

Chapters 5 through 10 describe most of the libraries contained within the Persistent Memory Development Kit (PMDK) and how to use them.

This chapter introduces the fundamental algorithms and data structures on which libpmemobj is built. After we first describe the overall architecture of the library, we discuss the individual components and the interaction between them that makes libpmemobj a cohesive system.

A Pool of Persistent Memory: High-level Architecture Overview

Figure 16-1 shows that libpmemobj comprises many isolated components that build on top of each other to provide a transactional object store.

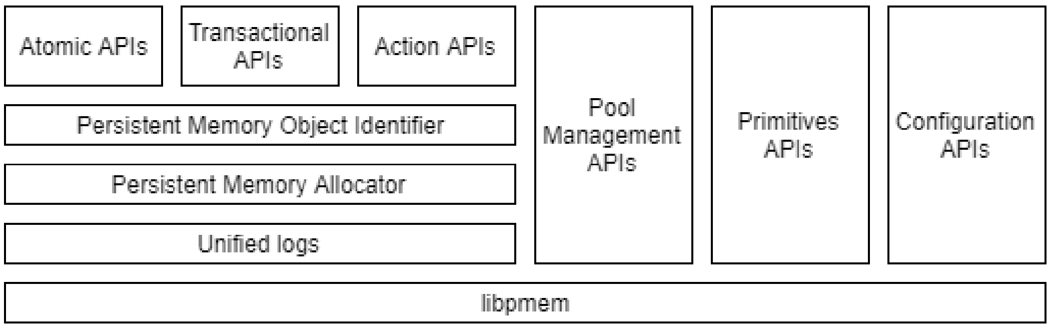

Figure 16-1. The modules of the libpmemobj architecture.

Everything is built on `libpmem` and its persistence primitives that the library uses to transfer data to persistent memory and persist it. Those primitives are also exposed through `libpmemobj`-specific APIs to applications that wish to perform low-level operations on persistent memory, such as manual cache flushing. Rather than relying on `libpmem`, these APIs are exposed so the high-level library can instrument, intercept, and augment all stores to persistent memory. This is useful for the instrumentation of runtime analysis tools, such as Valgrind `pmemcheck`, described in Chapter 12. More importantly, these functions are interception points for data replication, both local and remote.

Replication is implemented in a way that ensures all data written prior to calling drain will be safely stored in the replica as configured. (A drain operation is a barrier that waits for hardware buffers to complete their flush operation to ensure all writes have reached the media.) This works by initiating a write to the replica when a memory copy or a flush is performed, then waits for those writes to finish in the drain call. This mechanism guarantees the same behavior and ordering semantics for replicated and non-replicated pools.

On top of persistence primitives provided by `libpmem` is an abstraction for fail-safe modification of transactional called *unified logging.* The unified log is a single data structure and API for two different logging types used throughout `libpmemobj` to ensure fail-safety: transactions and atomic operations. This is one of the most crucial, performance-sensitive modules in the library because it is on the hot-path of almost every API. The unified log is a hybrid DRAM and persistent memory data structure accessed through a runtime context that organizes all memory operations that need to be performed within a single fail-safe atomic transaction and allows for critical performance optimizations.

The persistent memory allocator operates in the unified log context of either a transaction or a single atomic operation. This is the largest and most complex module in `libpmemobj` and is used to manage the potentially large amounts of persistent memory associated with the memory pool.

Each object stored in a persistent memory pool is represented by an object handle of type PMEMoid (persistent memory object identifiers). In practice, such a handle is a unique object identifier (OID) of global scope, which means that two objects from different pools will never have the same OID. An OID cannot be used as a direct pointer to an object. Each time the program attempts to read or write object data, it must obtain the current memory address of the object by converting its OID into a pointer. In contrast to the memory address, the OID value for a given object does not change during the life of an object, except for a `realloc()`, and remains valid after closing and reopening the pool. For this reason, if an object contains a reference to another persistent object, for example, to build a linked data structure, the reference must be an OID and not a memory address.

The atomic and transactional APIs are built using a combination of the persistent memory allocator and unified logs. The simplest public interface is the atomic API which runs a single allocator operation in a unified log context. That log context is not exposed externally and is created, initialized, and destroyed within a single function call.

The most general-purpose interface is the transactional API, which is based on a combination of undo logging for snapshots and redo logging for memory allocation and deallocation. This API has ACID (Atomicity, Consistency, Isolation, Durability) properties, and it is a relatively thin layer that combines the utility of unified logs and the persistent memory allocator.

For specific transactional use cases that need low-level access to the persistent memory allocator, there is an “action” API. The action API is essentially a passthrough to the raw memory allocator interface, alongside helpers for usability. This API can be leveraged to create low-overhead algorithms that issue fewer memory fences, as compared to general-purpose transactions, at the cost of ease of use.

All public interfaces produce and operate on PMEMoids as a replacement for pointers. This comes with space overhead because PMEMoids are 16 bytes. There is also a performance overhead for the translation to a normal pointer. The upside is that objects can be safely referenced between different instances of the application and even different persistent memory pools.

The pool management API opens, maps, and manages persistent memory resident files or devices. This is where the replication is configured, metadata and the heap are initialized, and all the runtime data is created. This is also where the crucial recovery of interrupted transactions happens. Once recovery is complete, all prior transactions are either committed or aborted, the persistent state is consistent, and the logs are clean and ready to be used again.

The Uncertainty of Memory Mapping: Persistent Memory Object Identifier

A key concept that is important for any persistent memory application is how to represent the relative position of an object within a pool of memory, and even beyond it. That is, how do you implement pointers? You could rely on normal pointers, which are relative to the beginning of the application's virtual address space, but that comes with many caveats. Using such pointers would be predicated on the pool of persistent memory always being located at the same place in the virtual address space of an application that maps it. This is difficult, if not impossible, to accomplish in a portable way on modern operating systems due to Address Space Layout Randomization (ASLR). Therefore, a general-purpose library for persistent memory programming must provide a specialized persistent pointer. Figure 16-2 shows a pointer from Object A to Object B. If the base address changes, the pointer no longer points to Object B.

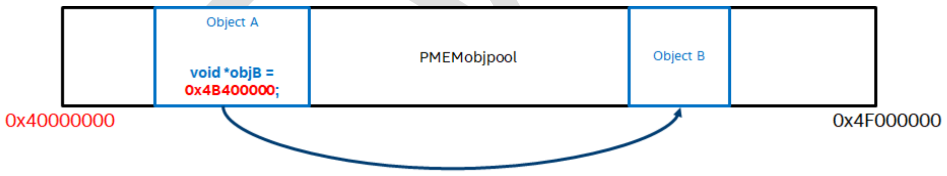

Figure 16-2. Example of using a normal pointer in a persistent memory pool.

An implementation of a general-purpose relative persistent pointer should satisfy these two basic requirements:

1. The pointer must remain valid across application restarts.
2. The pointer should unambiguously identify a memory location in the presence of many persistent memory pools, even if not located in a pool from which it was originally derived.

In addition to the previous requirements, you should also consider some potential performance problems:

- Additional space overhead over a traditional pointer. This is important because large fat pointers would take up more space in memory, and because fewer of these fat pointers would fit in a single CPU cache-line. This potentially increases the cost of operations in pointer-chasing heavy data structures, such as those found in B-Tree algorithms.
- The cost of translating persistent pointers to real pointers. Because dereferencing is an extremely common operation, this calculation must be as lightweight as possible and should involve as few instructions as possible. This is to ensure that persistent pointer usage is efficient and it doesn't generate too much code bloat during compilation.
- Preventing compiler optimizations through the dereferencing method. A complicated pointer translation might negatively impact the compiler's optimization passes. The translation method should ideally avoid operations that depend on an external state because that will prevent, among other things, auto-vectorization.

Satisfying the above requirements while maintaining low-overhead and C99 standard compliance is surprisingly difficult. We explored several options:

- 8-byte offset pointers, relative to the beginning of the pool, was quickly ruled out because it did not satisfy the second requirement and needed a pool base pointer to be provided to the translation method.
- 8-byte self-relative pointers, where the value of the pointer is the offset between the object's location and the pointer's location. This is potentially the fastest implementation because the translation method can be implemented as `` `ptr + (*ptr)` ``. However, this does not satisfy the second basic requirement. Additionally, it would require a special assignment method because the value of the pointer to the same object would differ depending on the pointer's location.

- 8-byte offset pointers with embedded memory pool identifier, which allows the library to satisfy the second requirement. This is an augmentation of the first method that additionally stores the identifier in the unused part of the pointer value by taking advantage of the fact that the usable size of the virtual address space is smaller than the size of the pointer on most modern CPUs. The problem with this method, however, is that the number of bits for the pool identifier is relatively small (16 bits on modern CPUs) and might shrink with future hardware.
- 16-byte fat offset pointer with pool identifier. This is the most obvious solution, which is similar to the one above but has 8 byte offset pointers and 8-byte pool identifiers. Fat pointers provide the best utility, at the cost of space overhead and some runtime performance.

`libpmemobj` uses the most generic approach of the 16-byte offset pointer. This allows you to make your own choice since all other pointer types can be directly derived from it. `libpmemobj` bindings for more expressive languages than C99, such as C++, can also provide different types of pointers with different tradeoffs.

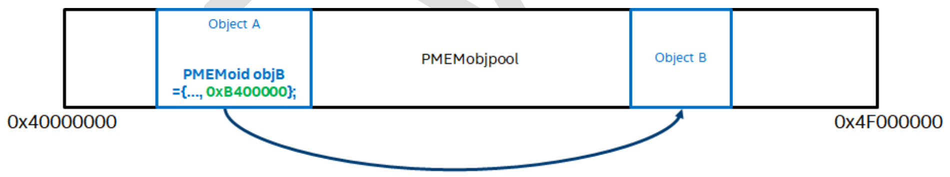

Figure 16-3. Example of using a PMEMoid in a persistent memory pool.

Figure 16-3 shows the translation method used to convert a `libpmemobj` persistent pointer, PMEMoid, into a valid C pointer. In principle, this approach is very simple. We lookup the base address of the pool through the pool identifier, then add the object offset to it. The method itself is static inline and defined in the public header file for `libpmemobj` to avoid a function call on every deference. The problem is the lookup method, which, for an application linked with a dynamic library, means a call to a different compilation unit, and that might be costly for a very commonly performed operation. To resolve this problem, the translation method has a per-thread cache of the last base address, which removes the necessity of calling the lookup with each dereferencing for the common case where persistent pointers from the same pool are accessed close together.

The pool lookup method itself is implemented using a radix tree that stores identifier-address pairs. This tree has a lock-free read operation, which is necessary because each non-cached pointer translation would otherwise have to acquire a lock to be thread-safe, and that would have a severe negative performance impact and could potentially serialize access to persistent memory.

Persistent Thread Local Storage: Using Lanes

Very early in the development of PMDK, we found that persistent memory programming close resembles multi-threaded programming because it requires restricting visibility of memory changes—either through locking or transactions—to other threads or instances of the program. But that is not the only similarity. The other similarity, which we discuss in this section, is how sometimes low-level code needs to store data that is unique to one thread of execution. Instead of associating data with a thread, it is associated with a transaction; and instead of being ephemeral, the data needs to be persistent.

In the case of `libpmemobj`, it needs a way to create an association between an in-flight transaction and its persistent logs. It also requires a way to reconnect to such logs after an unplanned interruption. The solution is to use a data structure called a "lane," which is simply a persistent byte buffer that is also transaction-local.

Lanes are limited in quantity, have a fixed size, and are located at the beginning of the pool. Each time a transaction starts, it chooses one of the lanes to operate from. Because there is a limited number of lanes, there is also a limited number of transactions that can run in parallel. For this reason, the size of the lane is relatively small, but the number of lanes is big enough as to be larger than a number of application threads that could feasibly run in parallel on current platforms and platforms coming in the foreseeable future.

The challenge of the lane mechanism is the selection algorithm; that is, which lane to choose for a specific transaction. It is a scheduler that assigns resources (lanes) to perform work (transactions).

The naïve algorithm, which was implemented in the earliest versions of `libpmemobj`, simply picked the first available lane from the pool. This approach has a few problems. First, the implementation of what effectively amounts to a single LIFO (last in, first out) data structure of lanes requires a lot of synchronization on the front of the stack, regardless of whether it is implemented as a linked list or an array, and thus reducing performance. The second problem is false sharing of lane data. False sharing occurs two or more threads operate on data that is being modified, causing CPU cache thrashing. And that is exactly what happens if multiple threads are continually fighting over the same number of lanes to start new transactions. The third problem is spreading the traffic across interleaved DIMMs. Interleaving is a technique that allows sequential traffic to take advantage of throughput of all of the DIMMs in the interleave set by spreading the physical memory across all available DIMMs. This is similar to striping (RAID0) across multiple disk drives. Depending on the size of the interleaved block, and the platform configuration, using naïve lane allocation might continuously use the same physical DIMMs, lowering the overall performance.

To alleviate these problems, the lane scheduling algorithm in `libpmemobj` is more complex. Instead of using a LIFO data structure, it uses an array of 8-byte spinlocks, one for each lane. Each thread is initially assigned a primary lane number, which is assigned in such a way as to minimize false sharing of both lane data and the spinlock array. The algorithm also tries to spread the lanes evenly across interleaved DIMMs. As long as there are fewer active threads than lanes, no thread will ever share a lane. When a thread attempts to start a transaction, it will try to acquire its primary lane spinlock, and if it is unsuccessful, it will try to acquire the next lane in the array.

The final lane scheduling algorithm decision took a considerable amount of research into various lane scheduling approaches. Compared to the naïve implementation, the current implementation has vastly improved performance, especially in heavily multi-threaded workloads.

Ensuring Power-Fail Atomicity: Redo and Undo Logging

The two fundamental concepts `libpmemobj` uses to ensure power fail safety are redo and undo logging. Redo logs are used to ensure atomicity of memory allocations, while undo logs are used to implement transactional snapshots. Before we discuss the many different possible implementation approaches, this section describes the basic ideas.

Transaction Redo Logging

Redo logging is a method by which a group of memory modifications that need to be done atomically are stored in a log and deferred until all modifications in the group are persistently stored. Once completed, the log is marked as complete and the memory modifications are processed (applied); the log can then be discarded. If the processing is interrupted before it finishes, the logging repeated until successful. Figure 16-4 shows the four phases of transaction redo logging.

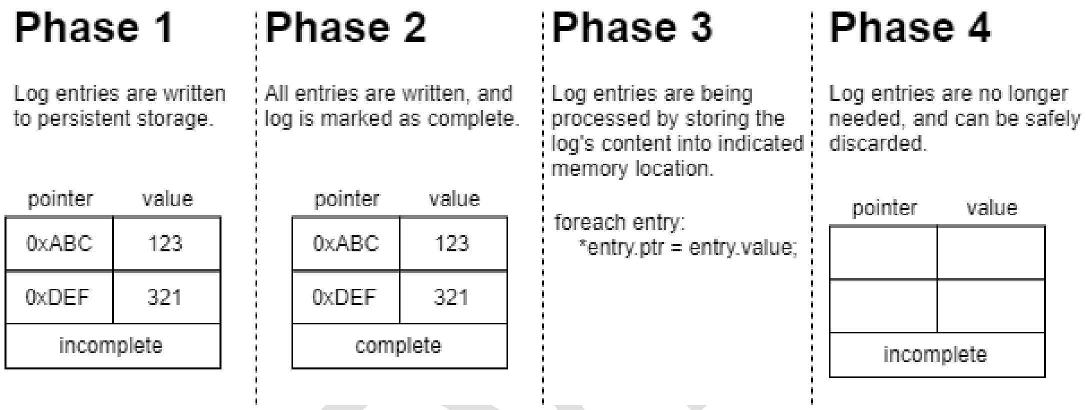
Phase 1
Log entries are written to persistent storage.
pointer
value
0xABC
123
0xDEF
321
incomplete
Phase 2
All entries are written, and log is marked as complete.
pointer
value
0xABC
123
0xDEF
321
complete
Phase 3
Log entries are being processed by storing the log's content into indicated memory location.
foreach entry:
*entry.ptr = entry.value;
Phase 4
Log entries are no longer needed, and can be safely discarded.
pointer
value
incomplete

Figure 16-4. The phases of a transaction redo log.

The benefit of this logging approach, in the context of persistent memory, is that all the log entries can be written and flushed to storage at once. An optimal implementation of redo logging uses only two synchronization barriers: once to mark the log as complete and once to discard it. The downside to this approach is that the memory modifications are not immediately visible, which makes for a more complicated programming model. Redo logging can sometimes be used alongside load/store instrumentation techniques which can redirect a memory operation to the logged location. However, this approach can be difficult to implement efficiently and is not well suited for a general-purpose library.

Transaction Undo Logging

Undo logging is a method by which each memory region of a group (undo transaction) that needs to be modified atomically is snapshotted into a log prior to the modification.

Once all memory modifications are complete, the log is discarded. If the transaction interrupted, the modifications in the log are rolled back. Figure 16-5 shows the three phases of the transaction undo logging.

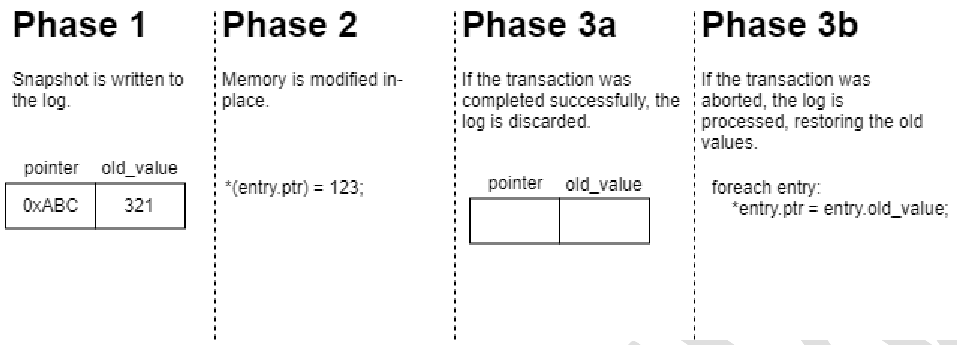
Phase 1
Snapshot is written to the log.
pointer
old_value
0xABC
321
Phase 2
Memory is modified in-place.
*(entry.ptr) = 123;
Phase 3a
If the transaction was completed successfully, the log is discarded.
pointer
old_value
Phase 3b
If the transaction was aborted, the log is processed, restoring the old values.
foreach entry:
*entry.ptr = entry.old_value;

Figure 16-5. Phases of a transaction undo log.

This type of log can have inferior performance characteristics because it requires a barrier for every snapshot that needs to be made, and the snapshotting itself must be fail-safe atomic, which presents its own challenges. An undo log benefit is that the changes are visible immediately, all, owing for a natural programming model.

The important observation here is that redo and undo logging are complimentary. Use redo logging for performance-critical code and where deferred modifications are not a problem; use undo logging where ease of use is important. This observation led to the current design of `libpmemobj` where a single transaction takes advantage of both algorithms.

Libpmemobj Unified Logging

Both redo and undo logging in `libpmemobj` share the same internal interface and data structure, which is called a unified log (or ulog for short). This is because redo and undo logging only differ in the execution order of the log phases. More precisely, when the log is applied on commit or recovery. In practice, however, there are performance considerations that require specialization in certain parts of the algorithm.

The ulog data structure contains one cache-line header with metadata and a variable length array of data bytes. The header consists of:

- A checksum for both the header and data, used only for redo logs.
- A monotonically increasing generation number of a transaction in the log, used only for undo logs.
- The total length in bytes of the data array.
- An offset of the next log in the group.

The last field is used to create a singly linked list of all logs that participate in a single transaction. This is because it is impossible to predict the total required size of the log at the beginning of the transaction, so the library cannot allocate a log structure that is the exact required length ahead of time. Instead, the logs are allocated on-demand and atomically linked into a list.

The unified log supports two ways of fail-safe inserting of entries:

1. **Bulk insert** takes an array of log entries, prepares the header of the log, and creates a checksum of both the header and data. Once done, a non-temporal copy, followed by a fence, is performed to store this structure into persistent memory. This is the way a group of deferred memory modifications forms a redo log with only one additional barrier at the end of the transaction. In this case, the checksum in the header is used to verify the consistency of the entire log. If that checksum doesn't match, the log is skipped during recovery.
2. **Buffer insert** takes only a single entry, checksums it together with the current generation number, and stores it in persistent memory through non-temporal stores followed by a fence. This method is used to create undo logs when snapshotting. Undo logs in a transaction are different than redo logs because during the commit's fast path they need to be invalidated instead of applied. The generation number is uses to maintain entries in the log buffer without having to overwrite the entire log space. Instead of laboriously writing zeros into the log buffer, the log is invalidated by incrementing the generation number. This works because the number is part of the data under checksum and changing it will cause the mismatch on the checksum. This algorithm allows `libpmemobj` to have only one additional fence for the transaction (on top of the fences needed for snapshots) to ensure fail-safety of a log, resulting in very low-overhead transactions.

Persistent Allocations: The Interface of a Transactional Persistent Allocator

The internal allocator interface in `libpmemobj` is far more complex than a typical volatile dynamic-memory allocator. First, it must ensure fail-safety of all its operations and cannot allow for any memory to become unreachable due to interruptions. Second, it must be transactional so that multiple operations on the heap can be done atomically alongside other modifications. And lastly, it must operate on the pool state, allocating memory from specific files instead of relying on the anonymous virtual memory provided by the operating system. All these factors contribute to an internal API that hardly resembles the standard `malloc()` and `free()`, shown in Listing 16-1.

Listing 16-1. The core persistent memory allocator interface that splits heap operations into two distinct steps.

```
int palloc_reserve(struct palloc_heap *heap, size_t size,...,
        struct pobj_action *act);
void palloc_publish(struct palloc_heap *heap,
        struct pobj_action *actv, size_t actvcnt,
        struct operation_context *ctx);
```

All memory operations, called “actions” in the API, are broken up into two individual steps.

The first step reserves the state that is needed to perform the operation. For allocations, this means retrieving a free memory block, marking it as reserved, and initializing the object’s content. This reservation is stored in a user-provided runtime variable. The library guarantees that if an application crashes while holding reservations, the persistent state is not affected. That is why these variables must not be persistent.

The second step is the act of exercising the reservations, which is called “publication.” Reservations can be published individually, but the true power of this API lies in its ability to group and publish many different actions together.

The internal allocator API also has a function to create an action that will set a memory location to a given value when published. This is used to modify the destination pointer value and is instrumental in making the atomic API of `libpmemobj` fail-safe.

All internal allocator APIs that need to perform fail-safe atomic actions take operation context as an argument, which is the runtime instance of a single log. It contains various state information, such as the total capacity of the log and the current number of entries. It exposes the functions to create either bulk or singular log entries. The allocator's function will log and then process all metadata modifications inside of the persistent log that belongs to the provided instance of the operating context.

Persistent Memory Heap Management: Allocator Design for Persistent Memory

The previous section described the interface for the memory allocation used internally in `libpmemobj`, but that was only the tip of the allocator iceberg. Before diving deeper into this topic, we briefly describe the principles behind normal volatile allocators so you can understand how persistent memory impacts the status quo.

Traditional allocators for volatile memory are responsible for efficient—in both time and space—management of operating system-provided memory pages. Precisely how this should be done for the generic case is an active research area of computer science; many different techniques can be used. All of them try to exploit the regularities in allocation and deallocation patterns to minimize heap fragmentation.

Most commonly-used general purpose memory allocators settled on an algorithm that we refer to as "segregated fit with page reuse and thread caching."

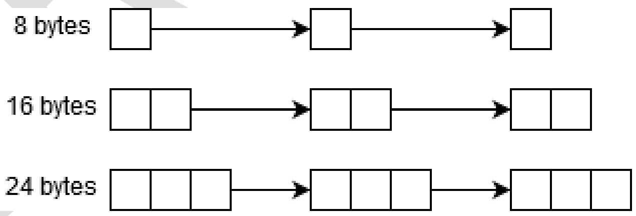

Figure 16-6. Example of free lists in a memory allocator.

This works by using a free list for many different sizes, shown in Figure 16-6, until some pre-defined threshold, after which it is sensible to allocate directly from the operating system. Those free lists are typically called bins or buckets and can be implemented in various ways, such as a simple linked list or contiguous buffer with boundary tags. Each incoming memory-allocation request is rounded-up to match one of the free lists, so there must be enough of them to minimize the amount of overprovisioned space for each allocation. This algorithm approximates a best-fit allocation policy that selects the memory block with the least amount of excess space for the request from the ones available.

Using this technique allows memory allocators to have average-case O(1) complexity while retaining the memory efficiency of best-fit. Another benefit is that rounding up of memory blocks and subsequent segregation forces some regularity to allocation patterns that otherwise might not exhibit any.

Some allocators also sort the available memory blocks by address and, if possible, allocate the one that is spatially collocated with previously selected blocks. This improves space efficiency by increasing the likelihood of reusing the same physical memory page. It also preserves temporal locality of allocated memory objects, which can minimize cache and translation lookaside buffer (TLB) misses.

One important advancement in memory allocators is scalability in multi-threaded applications. Most modern memory allocators implement some form of thread caching, where the vast majority of allocation requests are satisfied directly from memory that is exclusively assigned to a given thread. Only when memory assigned to a thread is entirely exhausted, or if the request is very large, the allocation will contend with other threads for operating system resources.

This allows for allocator implementations that have no locks of any kind, not even atomics, on the fast path. This can have a potentially significant impact on performance, even in the single-threaded case. This technique also prevents allocator-induced false-sharing between threads, since a thread will always allocate from its own region of memory. Additionally, the deallocation path often returns the memory block to the thread cache from which it originated, again preserving locality.

We mentioned earlier that volatile allocators manage operating system-provided pages but did not explain how they acquire those pages. This will become very important later as we discuss how things change for persistent memory. Memory is usually requested on-demand from the operating system either through `sbrk()`, which moves the break segment of the application, or anonymous `mmap()`, which creates new virtual memory mapping backed by the page cache. The actual physical memory is usually not assigned until the page is written to for the first time. When the allocator decides that it no longer needs a page, it can either completely remove the mapping using `unmap()` or it can tell the operating system to release the backing physical pages but keep the virtual mapping. This enables the allocator to reuse the same addresses later without having to memory map them again.

How does all of this translate into persistent memory allocators, and `libpmemobj` specifically?

The persistent heap must be resumable after application restart. This means that all state information must be either located on persistent memory or reconstructed on startup. If there are any active bookkeeping processes, those need to be restarted from the point at which they were interrupted. There cannot be any volatile state held in persistent memory, such as thread cache pointers. In fact, the allocator must not operate on any pointers at all because the virtual address of the heap can change between restarts.

In `libpmemobj`, the heap is rebuilt lazily and in stages. The entire available memory is divided into equally sized zones (except for the last one, which can be smaller than the others) with metadata at the beginning of each one. Each zone is subsequentially divided into variably sized memory blocks called chunks. Whenever there is an allocation request, and the runtime state indicates that there is no memory to satisfy it, the zone's metadata is processed, and the corresponding runtime state is initialized. This minimizes the startup time of the pool and amortizes the cost of rebuilding the heap state across many individual allocation requests.

There are three main reasons for having any runtime state at all. First, access latency of persistent memory can be higher than that of DRAM, potentially impacting performance of data structures placed on it. Second, separating runtime state from persistent state enables workflow where the memory is first reserved in runtime state, initialized, and only then the allocation is reflected on the persistent state. (This mechanism is described in the previous section.) Finally, maintaining fail-safety of complex persistent data structures is expensive and keeping them in DRAM allows the allocator to sidestep that cost.

The runtime allocation scheme employed by `libpmemobj` is segregated-fit with chunk reuse and thread caching as described above. Free lists in `libpmemobj`, called buckets, are placed in DRAM and are implemented as vectors of pointers to persistent memory blocks. The persistent representation of this data structure is a bitmap, located at the beginning of a larger buffer from which the smaller blocks are carved out. Such buffers in `libpmemobj` called runs, are variably sized and are allocated from the previously mentioned chunks. Very large allocations are directly allocated as chunks. Figure 16-7 shows the `libpmemobj` implementation.

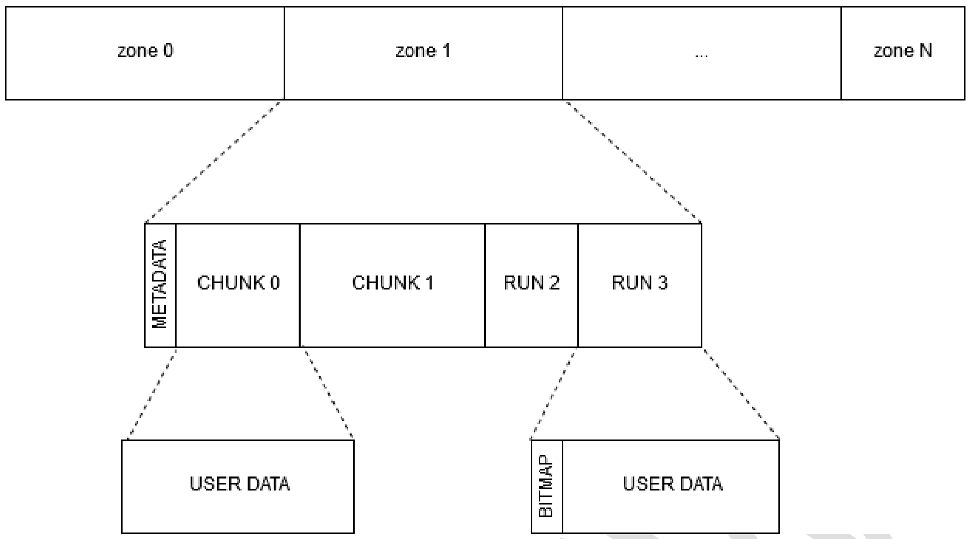

Figure 16-7. On-media layout of libpmemobj's heap.

Persistent allocators must also ensure consistency in the presence of failures. Otherwise, memory might become unreachable after an ungraceful shutdown of the application. One part of the solution is the API we outlined in the previous section. The other part is the careful design of the algorithms inside of the allocator that ensures no matter when the application is aborted, the state is consistent. This is also aided by redo logs, which are used to ensure atomicity of groups of non-contiguous persistent metadata changes.

One of the most impactful aspects of persistent memory allocation is how the memory is provisioned from the operating system. We previously explained that for normal volatile allocators, the memory is usually acquired through anonymous memory mappings that are backed by the page cache. In contrast, persistent heaps must use file-based memory mappings, backed directly by persistent memory. The difference might be subtle, but it has a significant impact on the way the allocator must be designed. The allocator must manually manage the entire virtual address space, retain information about any potential non-contiguous regions of the heap, and avoid excessive overprovisioning of virtual address space. Volatile allocators can rely on the operating system to coalesce non-contiguous physical pages into contiguous virtual ones, whereas persistent allocators cannot do the same without explicit and complicated techniques. Additionally, for some file system implementations, the allocator cannot assume that the physical memory is allocated at the time of the first page fault, so it must be conservative with internal block allocations.

Another problem for allocation from file-based mappings is that of perception. Normal allocators, due to memory over-commitment, seemingly never run out of memory because they are allocating the virtual address space, which is effectively infinite. There are negative performance consequences of address space bloat, and memory allocators actively try to avoid it, but they are not easily measurable in a typical application. In contrast, memory heaps allocate from a finite resource, the persistent memory device, or a file. This exacerbates the common phenomenon that is heap fragmentation by making it trivially measurable, creating the perception that persistent memory allocators are less efficient than volatile ones. They can be, but the operating system does a lot of work behind the scene to hide fragmentation of normal memory.

ACID Transactions: Efficient Low-Level Persistent Transactions

The four components we just described—lanes, redo logs, undo logs and the transactional memory allocator—form the basis of `libpmemobj's` implementation of ACID transactions that we defined in Chapter 4.

A transaction's persistent state consists of three logs. First is an undo log, which contains snapshots of user data. Second is an external redo log, which contains allocations and deallocations performed by the user. Third is an internal redo log, which is used to perform atomic metadata allocations and deallocations. This is technically not part of the transaction but is required to allocate the log extensions if they are needed. Without the internal redo log, it would be impossible to reserve and then publish a new log object in a transaction that already had user-made allocator actions in the external redo log.

All three logs have individual operation-context instances that are stored in runtime state of the lanes. This state is initialized when the pool is opened, and that is also when all the logs of the prior instance of the application are either processed or discarded. There is no special persistent variable that indicates whether past transactions in the log were successful or not. That information is directly derived from checksums stored in the log.

When a transaction begins, and it is not a nested transaction, it acquires a lane, which at that point must not contain any valid leftover logs. Runtime state of the transaction is stored in a thread-local variable, and that is where the lane variable is stored once acquired.

Transactional allocator operations use the external redo log and its associated operation context to call the appropriate reservation method which in turn creates an allocator action to be published at the time of transaction commit. The allocator actions are stored in a volatile array. If the transaction is aborted, all the actions are canceled, and the associated state is discarded. The complete redo log for memory allocations is created only at the time of publication. If the library is interrupted while creating the redo log, the next time the pool is opened, the checksum will not match and the transaction will be aborted by rolling back the undo log.

Transactional snapshots use the undo log and its context. The first time a snapshot is created, a new memory modification action is initialized. When published, that action creates an external redo log entry to increment the generation number of the associated undo log, invalidating its contents. This guarantees that if the external log is fully written and processed, it automatically discards the undo log, committing the entire transaction. If it is discarded, the undo log is processed, and the transaction is aborted.

To ensure that there are never two snapshots of the same memory location (this would be an inefficient use of space), there is a runtime range tree that is queried every time the application wants to create an undo log entry. If the new range overlaps with an existing snapshot, adjustments to the input arguments are made to avoid duplication. The same mechanism is also used to prevent snapshots of newly allocated data. Whenever new memory in a transaction is allocated, the reserved memory range is inserted into the ranges tree. Snapshotting new objects is redundant because they will be discarded automatically in the case of an abort.

To ensure that all memory modifications performed inside the transaction are durable on persistent memory once committed, the ranges tree is also used to iterate over all snapshots and call the appropriate flushing function on the modified memory locations.

Lazy Reinitialization of Variables: Storing the Volatile State on Persistent Memory

While developing software for persistent memory, it is often useful to store runtime (volatile) state inside of persistent memory locations. Keeping that state consistent, however, is extremely difficult, especially in multi-threaded applications.

The problem is the initialization of the runtime state. One solution is to simply iterate over all objects at the start of the application and initialize the volatile variables then, but that might significantly contribute to startup time of applications with large persistent pools. The other solution is to lazily reinitialize the variables on access. And that is what `libpmemobj` does for its built-in locks, but this also exposes this mechanism through an API for use with custom algorithms.

Lazy reinitialization of the volatile state is implemented using a lock-free algorithm that relies on a generation number stored alongside each volatile variable on persistent memory and inside the pool header. That number is increased by two every time a pool is opened. This means that a valid generation number is always even. When a volatile variable is accessed, its generation number is checked against the one stored in the pool header. If they match, it means that the object can be used and is simply returned to the application; otherwise, the object needs to be initialized before returning to ensure the initialization is thread-safe and is performed exactly once in a single instance of the application.

The naïve implementation could use a double-checked locking, where a thread would try to acquire a lock prior to initialization and verify again if the generation numbers match. If they still do not match, initialize the object and increase the number. To avoid the overhead that comes with using locks, the actual implementation first uses a compare and swap to set the generation number to a value that is equal to the generation number of the pool minus one, which is an odd number that indicates an operation is in progress. If this compare and swap were to fail, the algorithm would loop back to check if the generation number matches. If it is successful, however, the running thread initializes the variable and once again increments the generation number – this time to an even number that should match the number stored in the pool header.

Summary

This chapter describes the architecture and inner workings of `libpmemobj`. We also discuss the reasons for the choices that were made during the design and implementation of `libpmemobj`. With this knowledge, you can accurately reason about the semantics and performance characteristics of code written using this library.