CHAPTER 5

Introducing the Persistent Memory Development Kit

Previous chapters introduced the unique properties of persistent memory that make it special, and you are correct in thinking that writing software for such a novel technology is complicated. Anyone who has researched or developed code for persistent memory can testify to this. To make your job easier, Intel created the Persistent Memory Development Kit (PMDK). The team of PMDK developers envisioned it to be the standard library for all things persistent memory that would provide solutions to the common challenges of persistent memory programming.

Background

The PMDK has evolved to become a large collection of open source libraries and tools for application developers and system administrators to simplify managing and accessing persistent memory devices. Developed alongside evolving support for persistent memory in operating systems, it ensures that the libraries take advantage of all the features exposed through the operating system interfaces.

The PMDK libraries build on the SNIA NVM programming model (described in Chapter 3). They extend it to varying degrees, some by simply wrapping around the primitives exposed by the operating system with easy-to-use functions, others by providing complex data structures and algorithms for use with persistent memory. This means you are responsible for making an informed decision about which level of abstraction is the best for your use case.

Although the PMDK was created by Intel to support its hardware products, Intel is committed to ensuring the libraries and tools are both vendor- and platform-neutral.

This means that the PMDK is not tied to Intel® processors or Intel® persistent memory devices. It can be made to work on any other platform that exposes the necessary interfaces through the operating system, including Linux and Microsoft Windows*. We welcome and encourage contributions to PMDK from individuals, hardware vendors, and ISVs. The PMDK has a BSD 3-Clause License, allowing developers to embed it in any software, whether it's open source or proprietary. This allows you to pick and choose individual components of PMDK by integrating only the bits of code required.

The PMDK is available at no cost on GitHub (https://github.com/pmem/pmdk) and has a dedicated website at https://pmem.io. Man pages are delivered with PMDK and are available online under each library's own page. Appendix B of this book describes how to install it on your system.

An active persistent memory community is available through Google Forums at https://groups.google.com/forum/#!forum/pmem. This forum allows developers, system administrators, and others with an interest in persistent memory to ask questions and get assistance. This is a great resource.

Choosing the Right Semantics

With so many libraries available within the PMDK, it is important to carefully consider your options. The PMDK offers two library categories:

1. *Volatile* libraries are for use cases that only wish to exploit the capacity of persistent memory.
2. *Persistent* libraries are for use in software that wishes to implement fail-safe persistent memory algorithms.

While you are deciding how to best solve a problem, carefully consider which category it fits into. The challenges that fail-safe persistent programs present are significantly different from volatile ones. Choosing the right approach up front will minimize the risk of having to rewrite any code.

You may decide to use libraries from both categories for different parts of the application, depending on feature and functional requirements.

Volatile Libraries

Volatile libraries are typically simpler to use because they can fall back to dynamic random access memory (DRAM) when persistent memory is not available. This provides a more straightforward implementation. Depending on the workload, they may also have lower overall overhead compared to similar persistent libraries because they do not need to ensure consistency of data in the presence of failures.

This section explores the available libraries for volatile use cases in applications, including what the library is and when to use it. The libraries may have overlapping situation use cases.

libmemkind

What is it?

The memkind library, called `libmemkind`, is a user extensible heap manager built on top of `jemalloc`. It enables control of memory characteristics and partitioning of the heap between different *kinds* of memory. The kinds of memory are defined by operating system memory policies that have been applied to virtual address ranges. Memory characteristics supported by memkind without user extension include control of non-uniform memory access (NUMA) and page size features. The `jemalloc` non-standard interface has been extended to enable specialized *kinds* to make requests for virtual memory from the operating system through the memkind partition interface. Through the other memkind interfaces, you can control and extend memory partition features and allocate memory while selecting enabled features. The memkind interface allows you to create and control file-backed memory from persistent memory with PMEM kind.

Although `libmemkind` is not part of PMDK, it is maintained by the same group. Chapter 10 describes this library in more detail. You can download memkind and read the architecture specification and API documentation at http://memkind.github.io/memkind/. memkind is an open source project on GitHub at https://github.com/memkind/memkind.

When to use?

Choose `libmemkind` when you want to manually move select memory objects to persistent memory in a volatile application while retaining the traditional programming model. The memkind library provides familiar `malloc()` and `free()` semantics.

This is the recommended memory allocator for most volatile use cases of persistent memory.

Modern memory allocators usually rely on anonymous memory mapping to provision memory pages from the operating system. For most systems, this means that actual physical memory is allocated only when a page is first accessed, allowing the OS to overprovision virtual memory. Additionally, anonymous memory can be paged out if needed. When using memkind with file-based kinds, such as PMEM kind, while physical space is still only allocated on first access to a page, the other described techniques no longer apply. Memory allocation will fail when there is no memory available to be allocated, so it is important to handle such failures at the application level.

The described techniques also play an important role in hiding the inherent inefficiencies of manual dynamic memory allocation such as fragmentation, which causes allocation failures when not enough contiguous free space is available. Thus, file-based kinds can exhibit low space utilization for applications with irregular allocation/deallocation patterns. Such workloads may be better served with `libvmemcache`.

libvmemcache

What is it?

`libvmemcache` is an embeddable and lightweight in-memory caching solution that takes full advantage of large-capacity memory, such as persistent memory with direct memory access (DAX), through memory mapping in an efficient and scalable way. `libvmemcache` has unique characteristics:

- An extent-based memory allocator sidesteps the fragmentation problem that affects most in-memory databases and allows the cache to achieve very high space utilization for most workloads.
- The buffered least recently used (LRU) algorithm combines a traditional LRU doubly-linked list with a non-blocking ring buffer to deliver high degrees of scalability on modern multi-core CPUs.
- The `critnib` indexing structure delivers high performance while being very space efficient.

The cache is tuned to work optimally with relatively large value sizes. The smallest possible size is 256 bytes, but `libvmemcache` works best if the expected value sizes are above 1 kilobyte.

Chapter 10 describes this library in more detail. `libvmemcache` is an open source project on GitHub at https://github.com/pmem/vmemcache.

When to use?

Use `libvmemcache` when implementing caching for irregular workloads that typically would have low space efficiency when cached using a system with a normal memory allocation scheme.

libvmem

What is it?

`libvmem` is a deprecated predecessor to `libmemkind`. It is a `jemalloc`-derived memory allocator, with both metadata and objects allocations placed in file-based mapping. The `libvmem` library is an open source project available from https://pmem.io/pmdk/libvmem/.

When to use?

Use `libvmem` only if you have an existing application that uses `libvmem`, or if you need to have multiple completely separate heaps of memory. Otherwise, consider using `libmemkind`.

Persistent Libraries

Persistent libraries help applications maintain data structure consistency in the presence of failures. In contrast to the previously described volatile libraries, these provide new semantics and take full advantage of the unique possibilities enabled by persistent memory.

libpmem

What is it?

`libpmem` is a low-level C library that provides basic abstraction over the primitives exposed by the operating system. It automatically detects features available in the platform and chooses the right durability semantics and memory transfer (`memcpy()`) methods optimized for persistent memory. Most applications will need at least parts of this library.

Chapter 4 describes the requirements for applications using persistent memory, and Chapter 6 describes `libpmem` in more depth.

When to use?

Use `libpmem` when modifying an existing application that already uses memory-mapped I/O. Such applications can leverage the persistent memory synchronization primitives, such as user-space flushing, to replace `msync()`, thus reducing the kernel overhead.

Also use `libpmem` when you want to build everything from the ground up. It supports implementation of low-level persistent data structures with custom memory management and recovery logic.

libpmemobj

What is it?

`libpmemobj` is a C library that provides a transactional object store, with a manual dynamic memory allocator, transactions, and general facilities for persistent memory programming. This library solves many of the commonly encountered algorithmic and data structure problems when programming for persistent memory. Chapter 7 describes this library in detail.

When to use?

Use `libpmemobj` when the programming language of choice is C and when you need flexibility in terms of data structures design but can use a general-purpose memory allocator and transactions.

libpmemobj-cpp

What is it?

`libpmemobj-cpp`, also known as `libpmemobj++`, is a C++ header-only library that uses the metaprogramming features of C++ to provide a simpler, less error-prone interface to `libpmemobj`. It enables rapid development of persistent memory applications by reusing many concepts C++ programmers are already familiar with, such as smart pointers and closure-based transactions.

This library also ships with custom-made, STL-compatible data structures and containers, so that application developers do not have to reinvent the basic algorithms for persistent memory.

When to use?

When C++ is an option, `libpmemobj-cpp` is preferred for general-purpose persistent memory programming over `libpmemobj`. Chapter 7 describes this library in detail.

libpmemkv

What is it?

`libpmemkv` is a generic embedded local key-value store optimized for persistent memory. It is easy to use and ships with many different language integrations, including C, C++, and JavaScript.

This library has a pluggable backend for different storage engines. Thus, it can be used as a volatile library, although it was originally designed primarily to support persistent use cases.

Chapter 9 describes this library in detail.

When to use?

This library is the recommended starting point into the world of persistent memory programming because it is approachable and has a simple interface. Use it when complex and custom data structures are not needed, and a generic key-value store interface is enough to solve the current problem.

libpmemlog

What is it?

`libpmemlog` is a C library that implements a persistent memory append-only log file with power fail-safe operations.

When to use?

Use `libpmemlog` when your use case exactly fits into the provided log API; otherwise, a more generic library such as `libpmemobj` or `libpmemobj-cpp` might be more useful.

libpmemblk

What is it?

`libpmemblk` is a C library for managing fixed-size arrays of blocks. It provides fail-safe interfaces to update the blocks through buffer-based functions.

When to use?

Use `libpmemblk` only when a simple array of fixed blocks is needed and direct byte-level access to blocks is not required.

Tools and Command Utilities

PMDK comes with a wide variety of tools and utilities to assist in the development and deployment of persistent memory applications.

pmempool

What is it?

The `pmempool` utility is a tool for managing and offline analysis of persistent memory pools. Its variety of functionalities, useful throughout the entire lifecycle of an application, include:

- Obtaining information and statistics from a memory pool
- Checking a memory pool's consistency and repairing it if possible
- Creating memory pools
- Removing/deleting a previously created memory pool
- Updating internal metadata to the latest layout version
- Synchronizing replicas within a poolset
- Modifying internal data structures within a poolset
- Enabling or disabling pool and poolset features

When to use?

Use `pmempool` whenever you are creating or using an application based on any of the persistent libraries of PMDK.

pmemcheck

What is it?

The `pmemcheck` utility is a Valgrind-based tool for dynamic runtime analysis of common persistent memory errors, such as a missing flush or incorrect use of transactions. Chapter 12 describes this utility in detail.

When to use?

The `pmemcheck` utility is useful when developing an application using `libpmemobj`, `libpmemobj-cpp`, or `libpmem` because it can help you find bugs that are common in persistent applications. We suggest running error-checking tools early in the lifetime of a codebase to avoid a pile-up of hard-to-debug problems. The PMDK developers integrate `pmemcheck` tests into the continuous integration pipeline of PMDK, and we recommend the same for any persistent applications.

pmreorder

What is it?

The `pmreorder` utility helps detect data structure consistency problems of persistent applications in the presence of failures. It does this by first recording and then replaying the persistent state of the application while verifying consistency of the application's data structures at any possible intermediate state. Chapter 12 describes this utility in detail.

When to use?

Just like `pmemcheck`, `pmreorder` is an essential tool for finding hard-to-debug persistent problems and should be integrated into the development and testing cycle of any persistent memory application.

Summary

This chapter provides a brief listing of the libraries and tools available in PMDK and when to use them. You now have enough information to know what is possible. Throughout the rest of this book, you will learn how to create software using these libraries and tools.

The next chapter introduces `libpmem` and describes how to use it to create simple persistent applications.