CHAPTER 10

Volatile Use of Persistent Memory

Introduction

This chapter discusses how applications that require a large quantity of volatile memory can leverage high-capacity persistent memory as a complementary solution to dynamic random access memory (DRAM).

Applications that work with large datasets, like in-memory databases, caching systems, and scientific simulations, are often limited by the amount of volatile memory capacity available in the system or the cost of the DRAM required to load a complete data set. Persistent memory offers a tradeoff between price and performance.

In the memory-storage hierarchy (described in Chapter 1), data is stored in tiers with frequently accessed data placed in DRAM for low latency access, and less frequently accessed data is placed in larger capacity, higher latency storage devices. Examples of such solutions include Redis on Flash (https://redislabs.com/redis-enterprise/technology/redis-on-flash/) and Extstore for Memcached (https://memcached.org/blog/extstore-cloud/).

Compared with DRAM, persistent memory is relatively inexpensive and offers much higher capacity. Using these large capacities as volatile memory provides a new opportunity for memory-hungry applications that don't require persistence.

Using persistent memory as a volatile memory solution is advantageous when an application:

- Has control over data placement between DRAM and other storage tiers within the system
- Does not need to persist data

- Can use the native latencies of persistent memory, which may be slower than DRAM but are faster than non-volatile memory express (NVMe) NAND solid-state drives (SSDs).

Background

Applications manage different kinds of data structures such as user data, key-value stores, metadata, and working buffers. Architecting a solution that uses tiered memory and storage enhances application performance; for example, placing objects that are accessed frequently and require fast access in DRAM, while storing data that requires larger allocations and is not latency-sensitive on persistent memory in use as volatile memory.

Memory Allocation

As described in Chapters 1 through 3, persistent memory is exposed to the application using memory-mapped files on a persistent memory-aware file system that provides direct access to the application. Since `malloc()` and `free()` do not operate on files, an interface is needed that provides `malloc()` and `free()` semantics through an API for memory-mapped files as a source for memory allocation. This interface is implemented as the memkind library (http://memkind.github.io/memkind/).

How it Works

The memkind library is a user-extensible heap manager built on top of `jemalloc`, which enables control of memory characteristics and partitioning of the heap between *kinds* of memory. It was originally created to support different kinds of memory with the introduction of high bandwidth memory (HBM). A PMEM *kind* was introduced to support persistent memory.

Different "kinds" of memory are defined by the operating system memory policies that were 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 was extended so that specialized arenas could make requests for virtual memory from the operating system through the memkind partition interface. Through the other memkind interfaces, developers can control and extend memory partition features and allocate memory while selecting enabled features. Figure 10-1 shows an overview of libmemkind components and hardware support.

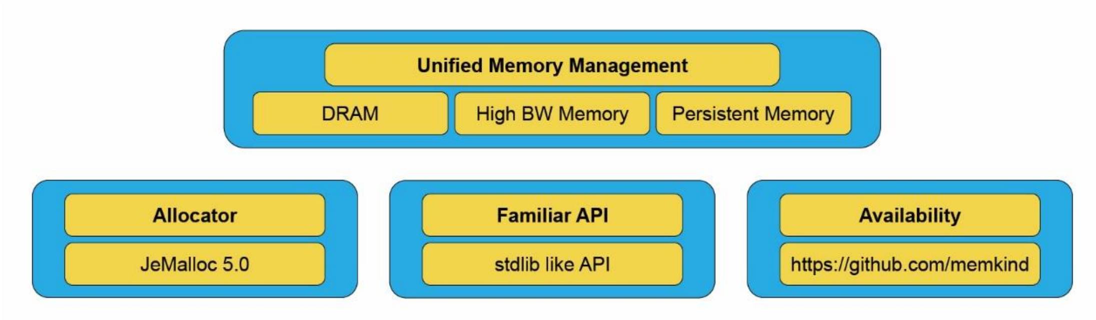

Figure 10-1. An overview of the memkind components and hardware support.

The memkind library serves as a wrapper that redirects memory allocation requests from an application to an allocator that manages the heap. At the time of publication, only the `jemalloc` allocator is supported. Future versions may introduce and support multiple allocators. Memkind provides `jemalloc` with different sources of memory: A *static kind* is created automatically, whereas a *dynamic kind* is created by an application using the `memkind_create_kind()` API.

The PMEM kind, which is dynamic, is best used with memory-addressable persistent storage through a DAX-enabled file system that supports load/store operations without being paged via the system page cache. The PMEM *kind* supports the traditional `malloc/free` interfaces on a memory-mapped file. A temporary file is automatically created on a mounted DAX file system and memory-mapped into the application's virtual address space. The temporary file is deleted when the program terminates, giving the perception of volatility.

Supported "Kinds" of Memory

When the *kind* of memory is `PMEM_KIND`, the memory allocation source is a memory-mapped file created as a temporary file on a persistent memory-aware file system.

For allocations from DRAM, the application sets the *kind* to `MEMKIND_DEFAULT` with the operating system's default page size. Refer to the memkind documentation for large and huge page support.

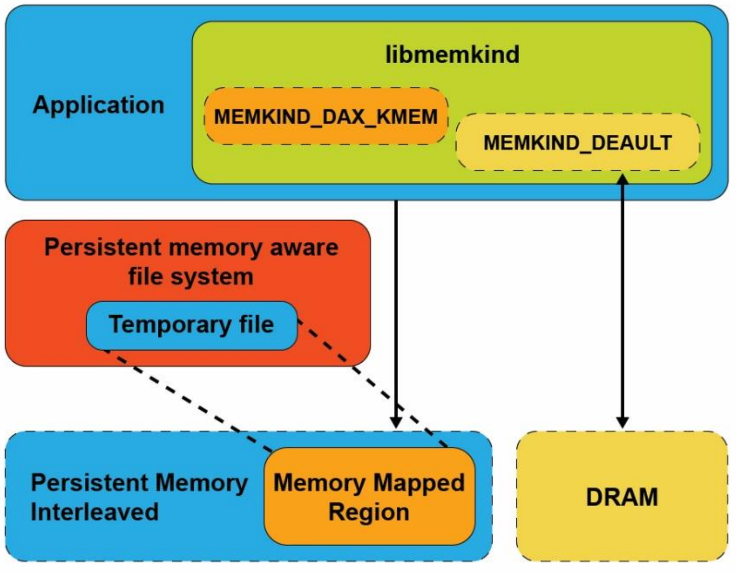

Figure 10.2. Application using different "kinds" of memory.

Figure 10-2 shows memory mappings from two memory sources: DRAM (`MEMKIND_DEFAULT`) and persistent memory (`PMEM_KIND`).

When using `PMEM_KIND`, the key points to understand are:

- Two pools of memory are available to the application from DRAM and persistent memory. Both can be accessed simultaneously by setting the kind type to `PMEM_KIND` and `MEMKIND_DEFAULT`.
- `jemalloc` is the single memory allocator used to manage all kinds of memory.
- memkind is a wrapper around `jemalloc` that provides a unified API for allocations from different kinds of memory.
- Memory allocations are provided by a temporary file created on a persistent memory-aware file system. The file is destroyed when the application exits.
- Allocations are not persistent.
- Using `libmemkind` for persistent memory requires simple modifications to the application.

The memkind API

The memkind API functions related to persistent memory programming are shown in Listing 10-1 and described in this section. The complete memkind API is available in the memkind man pages (http://memkind.github.io/memkind/man_pages/memkind.html).

Listing 10-1. Persistent memory related memkind API functions.

```
KIND CREATION MANAGEMENT:
int memkind_create_pmem(const char *dir, size_t max_size,
memkind_t *kind);
int memkind_create_pmem_with_config(struct memkind_config
*cfg, memkind_t *kind);
memkind_t memkind_detect_kind(void *ptr);
int memkind_destroy_kind(memkind_t kind);


KIND HEAP MANAGEMENT:
void *memkind_malloc(memkind_t kind, size_t size);
void *memkind_calloc(memkind_t kind, size_t num, size_t size);
void *memkind_realloc(memkind_t kind, void *ptr, size_t size);
void memkind_free(memkind_t kind, void *ptr);
size_t memkind_malloc_usable_size(memkind_t kind, void *ptr);
memkind_t memkind_detect_kind(void *ptr);

KIND CONFIGURATION MANAGEMENT:
struct memkind_config *memkind_config_new();
void memkind_config_delete(struct memkind_config *cfg);
void memkind_config_set_path(struct memkind_config *cfg, const
char *pmem_dir);
void memkind_config_set_size(struct memkind_config *cfg,
size_t pmem_size);
void memkind_config_set_memory_usage_policy(struct
memkind_config *cfg, memkind_mem_usage_policy policy);
```

Kind Management API

The memkind library supports a plugin architecture to incorporate new memory kinds, which are referred to as dynamic kinds. The memkind library provides the API to create and manage the heap for the new *kind*.

Kind Creation

Use the `memkind_create_pmem()` function to create a PMEM *kind* of memory from a file-backed source. This file is created as a `tmpfile(3)` in a specified directory and is unlinked so the filename is not listed under the directory and is automatically removed when the program terminates.

Use `memkind_create_pmem()` to create a fixed or dynamic heap size depending on the application requirement. Additionally, configurations can be created and supplied rather than passing in configuration options to the `*_create_*` function.

Creating a Fixed Size Heap

Applications that require a fixed amount of memory can specify a non-zero value for the `PMEM_MAX_SIZE` argument to `memkind_create_pmem()`. This defines the size of the memory pool to be created for the specified kind of memory. The value of `PMEM_MAX_SIZE` should be less than the available capacity of the file system specified in `PMEM_DIR` to avoid `ENOMEM` or `ENOSPC` errors. An internal data structure `struct memkind` is populated internally by the library and used by the memory management functions.

```
int memkind_create_pmem(PMEM_DIR, PMEM_MAX_SIZE, &pmem_kind)
```

The arguments to `memkind_create_pmem()` are:

- `PMEM_DIR` is the directory where the temp file is created
- `PMEM_MAX_SIZE` is the size of the memory region to be passed to `jemalloc`
- `&pmem_kind` is the address of a memkind data structure.

If successful, `memkind_create_pmem()` returns a value of zero. On failure, an error number is returned that `memkind_error_message()` can convert to an error message. Listing 10-2 shows how a 32MiB PMEM kind is created on a `/pmemfs` file system.

Listing 10-2. Creating a 32MiB PMEM kind.

```
#define PMEM_MAX_SIZE (1024 * 1024 * 32)

struct memkind *pmem_kind = NULL;
int err = 0;

// Create first PMEM partition with specific size
err = memkind_create_pmem("/pmemfs/", PMEM_MAX_SIZE,
&pmem_kind);
if (err) {
    print_err_message(err);
    return 1;
}
```

You can also create a heap with a specific configuration using the function `memkind_create_pmem_with_config()`. This function requires completing a `memkind_config` structure with optional parameters such as size, path to file, and memory usage policy. Listing 10-3 shows how to build a `test_cfg` using `meknind_config_new()`, then passing that configuration to `memkind_create_pmem_with_config()` to create a PMEM kind. We use the same `path` and `size` parameters from the Listing 10-2 example for comparison.

Listing 10-3. Creating PMEM kind with configuration.

```
struct memkind_config *test_cfg = memkind_config_new();
memkind_config_set_path(test_cfg, “/pmemfs/”);
memkind_config_set_size(test_cfg, 1024 * 1024 * 32);
memkind_config_set_memory_usage_policy(test_cfg,
MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE);

// create FPMEM partition with specific configuration
err = memkind_create_pmem_with_config(test_cfg, &pmem_kind);
if (err) {
    print_err_message(err);
    return 1;
```

```
}
```

Creating a Variable Size Heap

When `PMEM_MAX_SIZE` is set to zero, allocations are satisfied as long as the temporary file can grow. The maximum heap size growth is limited by the capacity of the file system mounted under the `PMEM_DIR` argument.

```
memkind_create_pmem(PMEM_DIR, 0, &pmem_kind)
```

The arguments to `memkind_create_pmem()` are:

- `PMEM_DIR` is the directory where the temp file is created
- `PMEM_MAX_SIZE` is 0
- `&pmem_kind` is the address of a memkind data structure

If the PMEM kind is created successfully, `memkind_create_pmem()` returns zero. On failure, `memkind_error_message()` can be used to convert an error number returned by `memkind_create_pmem()` to an error message.

Listing 10-4 shows how to create a PMEM kind with variable size.

Listing 10-4. Creating a PMEM kind with variable size.

```
struct memkind *pmem_kind = NULL;
int err = 0;
err = memkind_create_pmem("/pmemfs/",0,&pmem_kind);
if (err) {
    print_err_message(err);
    return 1;
}
```

Detecting the Memory Kind

memkind supports both automatic detection of a kind as well as a function to detect a kind associated with a memory referenced by a pointer.

Automatic Kind Detection

Support for automatically detecting the kind of memory was added to simplify code changes when adopting `libmemkind`. Thus, the memkind library will automatically retrieve the *kind* of memory pool where the allocation was done so that the heap management functions listed in Table 10-1 can be called without specifying the kind.

Table 10-1. Automatic kind detection functions and their equivalent specified kind functions and operations.

Operation	Memkind API with Kind	Memkind API using automatic detection
free	memkind_free(kind, ptr)	memkind_free(NULL, ptr)
realloc	memkind_realloc(kind, ptr, size)	memkind_realloc(NULL, ptr, size)
Get size of allocated memory	memkind_malloc_usable_size(kind, ptr)	memkind_malloc_usable_size(NULL, ptr)

The memkind library internally tracks the kind of a given object from the allocator metadata. However, to get this information some of the operations may need to acquire a lock to prevent accesses from other threads, which may negatively affect the performance in a multithreaded environment.

Memory Kind Detection API

Memkind also provides the `memkind_detect_kind()` function to query and return the kind of memory associated with the memory referenced by the pointer passed into the function. If the input pointer argument is `NULL`, it returns `NULL`. The input pointer argument that gets passed into `memkind_detect_kind()` must have been returned by a previous call to `memkind_malloc()`, `memkind_calloc()`, `memkind_realloc()` or `memkind_posix_memalign()`.

```
memkind_t memkind_detect_kind(void *ptr)
```

Similar to the automatic detection approach, this function has non-trivial performance overhead. Listing 10-5 shows how to detect the kind type.

Listing 10-5. pmem_detect_kind.c – how to automatically detect the 'kind' type.

```
/*
 * pmem_detect_kind.c - Uses the automatic 'kind'
 *                          detection API
 */

#include <memkind.h>

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

static char path[PATH_MAX]="/pmemfs/";

#define MALLOC_SIZE 512U
#define REALLOC_SIZE 2048U
#define ALLOC_LIMIT  1000U

static void *alloc_buffer[ALLOC_LIMIT];

static void print_err_message(int err)
{
    char error_message[MEMKIND_ERROR_MESSAGE_SIZE];
    memkind_error_message(err, error_message,
        MEMKIND_ERROR_MESSAGE_SIZE);
    fprintf(stderr, "%s\n", error_message);
}

static int allocate_pmem_and_default_kind(
    struct memkind *pmem_kind)
{
    unsigned i;
    for(i = 0; i < ALLOC_LIMIT; i++) {
        if (i%2)
            alloc_buffer[i] = memkind_malloc(
                pmem_kind, MALLOC_SIZE);
        else
            alloc_buffer[i] = memkind_malloc(
                MEMKIND_DEFAULT, MALLOC_SIZE);

        if (!alloc_buffer[i]) {
            return 1;
```

```
        }
    }

    return 0;
}

static int realloc_using_get_kind_only_on_pmem()
{
    unsigned i;
    for(i = 0; i < ALLOC_LIMIT; i++) {
        if (memkind_detect_kind(alloc_buffer[i]) !=
                MEMKIND_DEFAULT) {
            void *temp = memkind_realloc(NULL,
                alloc_buffer[i], REALLOC_SIZE);
            if (!temp) {
                return 1;
            }
            alloc_buffer[i] = temp;
        }
    }

    return 0;
}


static int verify_allocation_size(
    struct memkind *pmem_kind, size_t pmem_size)
{
    unsigned i;
    for(i = 0; i < ALLOC_LIMIT; i++) {
        void *val = alloc_buffer[i];
        if (i%2) {
            if (memkind_malloc_usable_size(pmem_kind,
                     val) != pmem_size ) {
                return 1;
            }
        } else {
            if (memkind_malloc_usable_size(
                MEMKIND_DEFAULT, val) != MALLOC_SIZE)
            {
                return 1;
            }
```

```
        }
    }

    return 0;
}

int main(int argc, char *argv[])
{

    struct memkind *pmem_kind = NULL;
    int err = 0;

    if (argc > 2) {
        fprintf(stderr,
            "Usage: %s [pmem_kind_dir_path]\n",
            argv[0]);
        return 1;
    } else if (argc == 2 && (realpath(argv[1], path)
            == NULL)) {
        fprintf(stderr,
            "Incorrect pmem_kind_dir_path %s\n",
            argv[1]);
        return 1;
    }

    fprintf(stdout,
            "This example shows how to distinguish "
            "allocation from different kinds using "
            "detect kind function"
            "\nPMEM kind directory: %s\n", path);

    err = memkind_create_pmem(path, 0, &pmem_kind);
    if (err) {
        print_err_message(err);
        return 1;
    }

    fprintf(stdout,
        "Allocate to PMEM and DEFAULT kind.\n");

    if (allocate_pmem_and_default_kind(pmem_kind)) {
        fprintf(stderr,
```

```
            "allocate_pmem_and_default_kind().\n");
        return 1;
    }

    if (verify_allocation_size(pmem_kind,
            MALLOC_SIZE)) {
        fprintf(stderr,
            "verify_allocation_size() before "
            "resize.\n");
        return 1;
    }

    fprintf(stdout,
        "Reallocate memory only on PMEM kind using "
        "memkind_detect_kind().\n");

    if (realloc_using_get_kind_only_on_pmem()) {
        fprintf(stderr,
            "realloc_using_get_kind_only_on_pmem()."
            "\n");
        return 1;
    }

    if (verify_allocation_size(pmem_kind,
            REALLOC_SIZE)) {
        fprintf(stderr,
            "verify_allocation_size() after resize."
            "\n");
        return 1;
    }

    err = memkind_destroy_kind(pmem_kind);
    if (err) {
        print_err_message(err);
        return 1;
    }

    fprintf(stdout, "Memory from PMEM kind was "
        "successfully reallocated.\n");

    return 0;
}
```

Destroying Kind Objects

Use the `memkind_destroy_kind()` function to delete the kind object that was previously created using the `memkind_create_pmem()` or `memkind_create_pmem_with_config()` function. The `memkind_destroy_kind()` is defined as:

```
int memkind_destroy_kind(memkind_t kind);
```

Using the same `pmem_detect_kind.c` code from Listing 10-5, Listing 10-6 shows how the kind is destroyed before the program exits.

Listing 10-6. Destroying a kind object.

```
    err = memkind_destroy_kind(pmem_kind);
    if (err) {
        print_err_message(err);
        return 1;
    }

    fprintf(stdout, "Memory from PMEM kind was "
        "successfully reallocated.\n");

    return 0;
```

When the kind returned by `memkind_create_pmem()` or `memkind_create_pmem_with_config()` is successfully destroyed, all the allocated memory for the kind object is freed.

Heap Management API

The heap management functions described in this section have an interface modeled on the ISO C standard API, with an additional “kind” parameter to specify the memory type used for allocation.

Allocating Memory

The memkind library provides `memkind_malloc()`, `memkind_calloc()` and `memkind_realloc()` functions for allocating memory, defined as follows:

```
void *memkind_malloc(memkind_t kind, size_t size);
void *memkind_calloc(memkind_t kind, size_t num, size_t size);
void *memkind_realloc(memkind_t kind, void *ptr, size_t size);
```

`memkind_malloc()` allocates size bytes of uninitialized memory of the specified kind. The allocated space is suitably aligned (after possible pointer coercion) for storage of any object type. If `size` is 0, then `memkind_malloc()` returns `NULL`.

`memkind_calloc()` allocates space for `num` objects, each are `size` bytes in length. The result is identical to calling `memkind_malloc()` with an argument of `num * size`. The exception is that the allocated memory is explicitly initialized to zero bytes. If `num` or `size` is 0, then `memkind_calloc()` returns `NULL`.

`memkind_realloc()` changes the size of the previously allocated memory referenced by `ptr` to size bytes of the specified kind. The contents of the memory remain unchanged, up to the lesser of the new and old sizes. If the new size is larger, the contents of the newly allocated portion of the memory are undefined. If successful, the memory referenced by `ptr` is freed and a pointer to the newly allocated memory is returned.

The examples in Listing 10-7 show how to allocate memory from DRAM and persistent memory (pmem_kind) using `memkind_malloc()`. Rather than using the common C library `malloc()` for DRAM memory and `memkind_malloc()` for persistent memory, we recommend using a single library to simplify code.

Listing 10-7. An example of allocating memory from both DRAM and persistent memory. .

```
/*
 * Allocates 100 bytes using appropriate "kind"
 * of volatile memory
 */


// Create first PMEM partition with a specific size
```

```
  err = memkind_create_pmem(path, PMEM_MAX_SIZE, &pmem_kind);

  if (err) {

      print_err_message(err);

      return 1;

  }

  char *pstring = memkind_malloc(pmem_kind, 100);
  char *dstring = memkind_malloc(MEMKIND_DEFAULT, 100);
```

Freeing Allocated Memory

To avoid memory leaks, allocated memory can be freed using the `memkind_free()` function, defined as:

```
void memkind_free(memkind_t kind, void *ptr);
```

`memkind_free()` causes the allocated memory referenced by `ptr` to be made available for future allocations. This pointer must be returned by a previous call to `memkind_malloc()`, `memkind_calloc()`, `memkind_realloc()` or `memkind_posix_memalign()`. Otherwise, if `memkind_free(kind, ptr)` was previously called, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed. In cases where the `kind` is unknown in the context of the call to `memkind_free()`, `NULL` can be given as the kind specified to `memkind_free()`, but this will require an internal lookup for the correct kind. Always specify the correct kind because the lookup for kind could result in serious performance penalty.

Listing 10-8 shows four examples of `memkind_free()` being used. The first two specify the kind, and the second two use `NULL`.

Listing 10-8. Examples of memkind_free() usage.

```
/* Free the memory by specifying the 'kind' */
memkind_free(MEMKIND_DEFAULT, dstring);
memkind_free(PMEM_KIND, pstring);

/* Free the memory using automatic 'kind' detection */
memkind_free(NULL, dstring);
```

```
memkind_free(NULL, pstring);
```

Kind Configuration Management

Memory Usage Policy

A tunable run time option set by the `dirty_decay_ms` in `jemalloc` determines how fast it returns unused memory back to the operating system. A shorter decay time purges unused memory pages faster but the purging costs CPU cycles. Trade-offs between memory and CPU cycles needs to be carefully thought out before setting this parameter.

A new implementation was introduced in memkind release v1.9 to improve memory utilization and reduce fragmentation. The first implementation supports two policies:

1. MEMKIND_MEM_USAGE_POLICY_DEFAULT
2. MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE

The minimum and maximum values for `dirty_decay_ms` using the `MEMKIND_MEM_USAGE_POLICY_DEFAULT` are 0ms to 10,000ms for arenas assigned to a PMEM kind. Setting `MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE` sets shorter decay times to purge unused memory faster, resulting in reducing memory usage. To define the memory usage policy, use `memkind_config_set_memory_usage_policy()`, defined below:

```
void memkind_config_set_memory_usage_policy (struct
memkind_config *cfg, memkind_mem_usage_policy policy );
```

`MEMKIND_MEM_USAGE_POLICY_DEFAULT` is the default memory usage policy.
`MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE` allows changing the `dirty_decay_ms` parameter.

Listing 10-9 shows how to use `memkind_config_set_memory_usage_policy()` with a custom configuration.

Listing 10-9. An example of a custom configuration and memory policy use.

```
/*
 * pmem_config.c - Demonstrates the use of several
 *             configuration functions within
 *             libmemkind.
 */

#include <memkind.h>

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

#define PMEM_MAX_SIZE (1024 * 1024 * 32)

static char path[PATH_MAX] = "pmemfs//";

static void print_err_message(int err)
{
    char error_message[MEMKIND_ERROR_MESSAGE_SIZE];
    memkind_error_message(err, error_message,
        MEMKIND_ERROR_MESSAGE_SIZE);
    fprintf(stderr, "%s\n", error_message);
}

int main(int argc, char *argv[])
{
    struct memkind *pmem_kind = NULL;
    int err = 0;

    if (argc > 2) {
        fprintf(stderr,
            "Usage: %s [pmem_kind_dir_path]\n",
            argv[0]);
        return 1;
    } else if (argc == 2 &&
        (realpath(argv[1], path) == NULL)) {
        fprintf(stderr,
            "Incorrect pmem_kind_dir_path %s\n",
            argv[1]);
        return 1;
    }
```

```

    fprintf(stdout,
            "This example shows how to use custom "
            "configuration to create pmem kind."
            "\nPMEM kind directory: %s\n", path);

    struct memkind_config *test_cfg =
        memkind_config_new();
    if (!test_cfg) {
        fprintf(stderr,
            "Unable to create memkind cfg.\n");
        return 1;
    }

    memkind_config_set_path(test_cfg, path);
    memkind_config_set_size(test_cfg, PMEM_MAX_SIZE);
    memkind_config_set_memory_usage_policy(test_cfg,
        MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE);


    // Create PMEM partition with the configuration
    err =  memkind_create_pmem_with_config(test_cfg,
        &pmem_kind);
    if (err) {
        print_err_message(err);
        return 1;
    }

    err = memkind_destroy_kind(pmem_kind);
    if (err) {
        print_err_message(err);
        return 1;
    }

    memkind_config_delete(test_cfg);

    fprintf(stdout,
        "PMEM kind and configuration was successfully"
        " created and destroyed.\n");

    return 0;
}
```

Additional memkind Code Examples

Table 10-2 lists the code examples available on GitHub at https://github.com/memkind/memkind/tree/master/examples.

Table 10-2. Source code examples using libmemkind.

File Name	Description
pmem_kinds.c	Creating and destroying PMEM kind with defined or unlimited size.
pmem_malloc.c	Allocating memory and the possibility to exceed PMEM kind size.
pmem_malloc_unlimited.c	Allocating memory with unlimited kind size.
pmem_usable_size.c	Viewing the difference between the expected and the actual allocation size.
pmem_alignment.c	Using memkind alignment and how it affects allocations.
pmem_multithreads.c	Using multithreading with independent PMEM kinds.
pmem_multithreads_onekind.c	Using multithreading with one main PMEM kind.
pmem_and_default_kind.c	Allocating in standard memory and file-backed memory (PMEM kind).
pmem_detect_kind.c:	Distinguishing allocation from different kinds using the detect kind function.
pmem_config.c	Using custom configuration to create PMEM kind.
pmem_free_with_unknown_kind.c	Allocating in-standard memory, file-backed memory (PMEM kind), and free memory without needing to remember which kind it belongs to.
pmem_cpp_allocator.cpp	Shows usage of C++ allocator mechanism designed for file-backed memory kind with different data structures like vector, list, and map.

Expanding Volatile Memory Using Persistent Memory

Persistent memory is treated by the kernel as a device. In a typical usage, a persistent memory-aware file system is created, and files are memory-mapped into the virtual address space of a process to give applications direct load/store access to persistent memory regions.

A new feature was added to Linux* kernel v5.1 so that persistent memory can be used more broadly as RAM. This is done by binding a persistent memory device to the kernel, and the kernel manages it as DRAM. Since persistent memory has different characteristics than DRAM, memory provided by this device is visible as a separate NUMA node on its corresponding socket.

To programmatically allocate memory from a NUMA node created for persistent memory, a new static kind, called MEMKIND_DAX_KMEM, was added to `libmemkind`.

```
memkind_malloc(MEMKIND_DAX_KMEM, size_t size)
```

Using `MEMKIND_DAX_KMEM`, you can use both DRAM and persistent memory as separate NUMA nodes in a single application, similar to the logic used with file-based `PMEM_KIND`. Figure 10-3 shows an application that created two static kind objects: `MEMKIND_DEFAULT` and `MEMKIND_DAX_PMEM`.

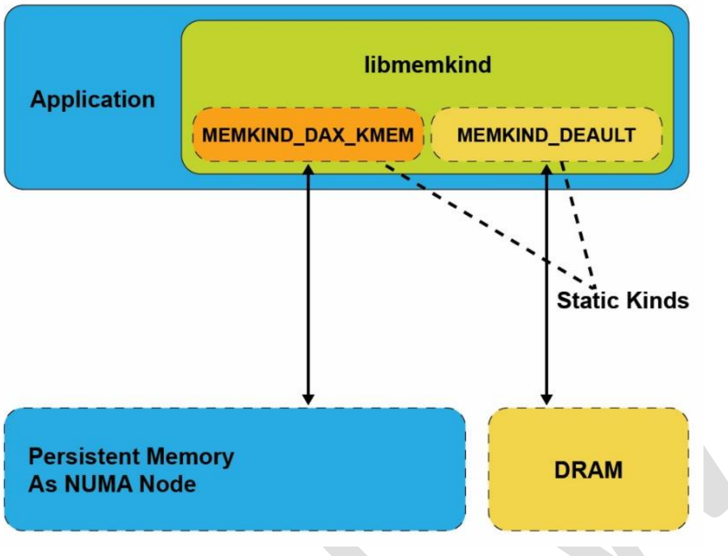

Figure 10-3. An application that created two kind objects from different types of memory.

The difference between the two kinds of memory-mapped to the application in Figure 10-3 is that `MEMKIND_DAX_KMEM` uses a memory-mapped file with the `MAP_PRIVATE` flag, while the dynamic `MEMKIND_DEFAULT` created with `memkind_create_kind()` uses `MAP_SHARED` when memory-mapping files on a DAX-enabled file system. The MAP_SHARED and MAP_PRIVATE definitions from the mmap() system call are defined in the man pages as follows:

MAP_SHARED

Share this mapping. Updates to the mapping are visible to other processes mapping the same region and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of `msync(2)`.)

MAP_PRIVATE

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file and are not carried through to the underlying file. It is unspecified whether changes made to the file after the `mmap()` call are visible in the mapped region.

Child processes created using the `fork(2)` system call inherit the same `MAP_PRIVATE` mappings from the parent process. When memory pages are modified by the parent process, a copy-on-write mechanism is triggered by the kernel to create an unmodified copy for child process. These pages are allocated on the same NUMA node as the original page.

C++ Allocator for PMEM Kind

To enable C++ developers to allocate from a PMEM kind of memory, the `pmem::allocator` class template, which conforms to C++11 allocator requirements, was developed. It can be used with C++ compliant data structures from:

- Standard Template Library (STL)
- Intel® Threading Building Blocks (Intel® TBB) library

The `pmem::allocator` class template uses the `memkind_create_pmem()` function described previously. This allocator is stateful and has no default constructor. Table 10-3 describes the available allocator methods.

Table 10-3. pmem::allocator methods.

pmem::allocator(const char *dir, size_t max_size)
pmem::allocator(const std::string& dir, size_t max_size)
template <typename U> pmem::allocator<T>::allocator(const pmem::allocator<U>&)
template <typename U> pmem::allocator(allocator<U>&& other)
pmem::allocator<T>::~allocator()

T* pmem::allocator<T>::allocate(std::size_t n) const
void pmem::allocator<T>::deallocate(T* p, std::size_t n) const
template <class U, class... Args> void pmem::allocator<T>::construct(U* p, Args... args) const
void pmem::allocator<T>::destroy(T* p) const

For more information about the `pmem::allocator` class template, refer to the `pmem allocator(3)` man page.

Nested Containers

Challenges occur while working with multilevel containers such as a vector of sets of lists, tuples, maps, strings, and so on. When the outermost container is constructed, an instance of `pmem::allocator` is passed as a parameter to the constructor. How should you handle nested objects stored in the outermost container?

Imagine you need to create a vector of strings and store it in persistent memory. The challenges—and their solutions—for this task include:

1. You cannot use `std::string` for this purpose because it is an alias of the `std::basic_string`. The `std::allocator` requires a new alias that uses `pmem:allocator`.

   Solution: A new alias called `pmem_string` is defined as a `typedef` of `std::basic_string` when created with `pmem::allocator`.

2. How to ensure that an outermost vector will properly construct nested `pmem_string` with a proper instance of `pmem::allocator`.

   Solution: From C++11 and later, the `std::scoped_allocator_adaptor` class template can be used with multilevel containers. The purpose of this adaptor is to correctly initialize stateful allocators in nested containers, such as when all levels of a nested container must be placed in the same memory segment.

C++ Examples

This section presents several full-code examples demonstrating the use of `libmemkind` using C and C++.

Using the pmem::allocator

As mentioned earlier, you can use `pmem::allocator` with any STL-like data structure. The code sample in Listing 10-10 includes a `pmem_allocator.h` header file to use `pmem::allocator`.

Listing 10-10. Using pmem::allocator with std:vector.

```
/*
 * pmem_allocator.cpp - Demonstrates using the
 *                      pmem::allocator
 *                              with std:vector.
 */

#include <pmem_allocator.h>
#include <vector>
#include <cassert>

int main(int argc, char *argv[]) {
    const size_t pmem_max_size = 64*1024*1024; //64 MB
    const std::string pmem_dir("/pmemfs/");

```

```
47      // Create allocator object
48      libmemkind::pmem::allocator<int> alc(pmem_dir,
49      pmem_max_size);
50      // Create std::vector with our allocator.
51      std::vector<int, libmemkind::pmem::allocator<int>
52      > v(alc);
53
54      for(int i = 0; i < 100; ++i)
55          v.push_back(i);
56
57      for(int i = 0; i < 100; ++i)
58          assert(v[i] == i);
59
60      return 0;
61  }
```

- Line 43: We define a persistent memory mapping of 64MiB.
- Line 48: We create an allocator object `alc` of type `pmem::allocator<int>`.
- Line 51: We create a vector object `v` of type `std::vector<int, pmem::allocator<int> >` and pass in the `alc` from line 48 object as an argument. The `pmem::allocator` is stateful and has no default constructor. This requires passing the allocator object to the vector constructor; otherwise, a compilation error occurs if the default constructor of `std::vector<int, pmem::allocator<int> >` is called because the vector constructor will try to call the default constructor of `pmem::allocator`, which does not exist yet.

Creating a Vector of Strings

Listing 10-11 shows how to create a vector of strings that resides in persistent memory. We define `pmem_string` as a `typedef` of `std::basic_string` with `pmem::allocator`. In this example, `std::scoped_allocator_adaptor` allows the vector to propagate the `pmem::allocator` instance to all `pmem_string` objects stored in the vector object.

Listing 10-11. Creating a vector of strings.

```
/*
 * vector_of_strings.cpp - Demonstrated how to create
 *                   a vector of strings residing on
 *                   persistent memory.
 */

#include <pmem_allocator.h>
#include <vector>
#include <string>
#include <scoped_allocator>
#include <cassert>
#include <iostream>

typedef libmemkind::pmem::allocator<char> str_alloc_type;

typedef std::basic_string<char, std::char_traits<char>, str_alloc_type> pmem_string;

typedef libmemkind::pmem::allocator<pmem_string> vec_alloc_type;

typedef std::vector<pmem_string, std::scoped_allocator_adaptor<vec_alloc_type> > vector_type;

int main(int argc, char *argv[]) {
    const size_t pmem_max_size = 64*1024*1024; //64 MB
    const std::string pmem_dir("/tmp");

    // Create allocator object
    vec_alloc_type alc(pmem_dir, pmem_max_size);
    // Create std::vector with our allocator.
    vector_type v(alc);

    v.emplace_back("Foo");
    v.emplace_back("Bar");

    for(auto str : v) {
            std::cout << str << std::endl;
    }

```

```
70        return 0;
71    }
```

- Line 48: We define `pmem_string` as a `typedef` of `std::basic_string`.
- Line 50: We define the `pmem::allocator` using the `pmem_string` type.
- Line 52: Using `std::scoped_allocator_adaptor` allows the `vector` to propagate the `pmem::allocator` instance to all `pmem_string` objects stored in the `vector` object.

See more examples in the memkind examples directory on GitHub (https://github.com/memkind/memkind/tree/master/examples).

libvmemcache: An Efficient Volatile Key-Value Cache for Large-Capacity Persistent Memory

Some existing in-memory databases (IMDB) rely on manual dynamic memory allocations (`malloc`, `jemalloc`, `tcmalloc`), which can exhibit memory fragmentation (external and internal) when run for a long period leaving large amounts of memory un-allocatable. Internal and external fragmentation is briefly explained as follows:

- *Internal fragmentation* occurs when more than the needed memory is allocated, and the unused memory is contained within the allocated region. For example, if the requested allocation size is 200 bytes, a chunk of 256 bytes is allocated.

- *External fragmentation* occurs when variable memory sizes are allocated dynamically, resulting in a failure to allocate a contiguous chunk of memory, although the requested chunk of memory remains available in the system. This problem is more pronounced when large capacities of persistent memory are being used as volatile memory. Applications with substantially long runtimes need to resolve this problem, especially if the allocated sizes have considerable variation. Applications and runtime environments handle this problem in different ways:

- Java and .NET use compacting garbage collection
- Redis and Apache Ignite* use defragmentation algorithms
- Memcached uses a slab allocator

Each of the above allocator mechanisms has pros and cons. Garbage and defragmentation algorithms require processing to occur on the heap to free unused allocations or move data to create contiguous space. Slab allocators usually define a fixed set of different sized buckets at initialization without knowing how many of each bucket the application will need. If the slab allocator depletes a certain bucket size, it allocates from larger sized buckets, which reduces the amount of free space. These three mechanisms can potentially block the application's processing and reduce its performance.

libvmemcache Overview

libvmemcache is an embeddable and lightweight in-memory caching solution with a key-value store at its core. It is designed to take full advantage of large-capacity memory, such as persistent memory, efficiently using memory mapping in a scalable way. It is optimized for use with memory-addressable persistent storage through a DAX-enabled file system that supports load/store operations. libvmemcache has these unique characteristics:

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

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

libvmemcache has more control over the allocation because it implements a custom memory-allocation scheme using an extents-based approach (like that of file system extents). libvmemcache can, therefore, concatenate and achieve substantial space efficiency. Additionally, because it is a cache, it can evict data to allocate new entries in a worst-case scenario. libvmemcache will *always* allocate exactly as much memory as it freed, minus metadata overhead. This is not true for caches based on common memory allocators such as memkind. libvmemcache is designed to work with terabyte-sized in-memory workloads, with very high space utilization.

libvmemcache works by automatically creating a temporary file on a DAX-enabled file system and memory-mapping it into the application's virtual address space. The temporary file is deleted when the program terminates and gives the perception of volatility. Figure 10-4 shows the application using traditional `malloc()` to allocate memory from DRAM and using libvmemcache to memory map a temporary file residing on a DAX-enabled file system from persistent memory.

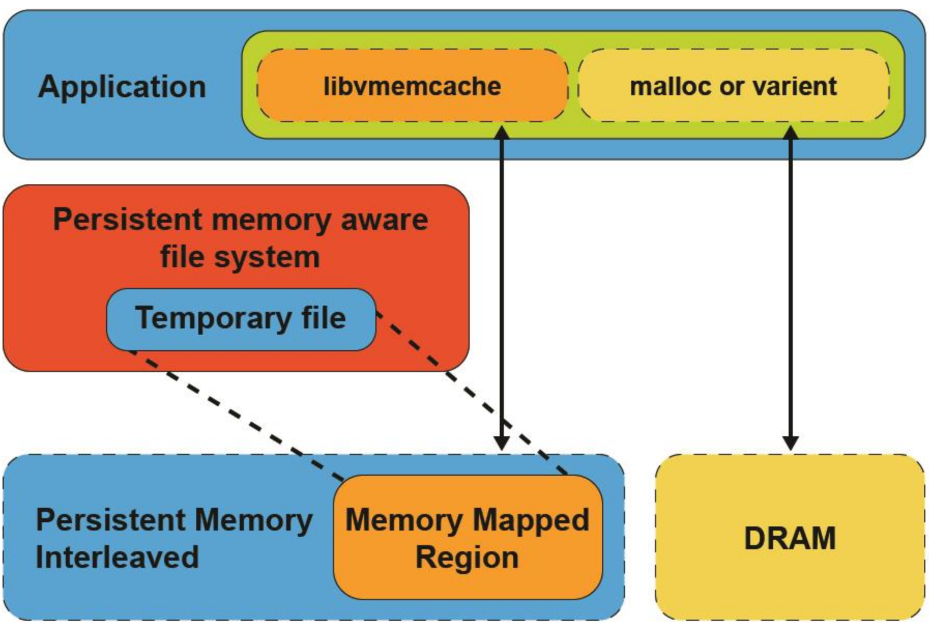

Figure 10-4. An application using libvmemcache memory maps a temporary file from a DAX-enabled file system.

Although libmemkind supports different kinds of memory and memory consumption policies, the underlying allocator is `jemalloc`, which uses dynamic memory allocation. Table 10-4 compares the implementation details of libvmemcache and libmemkind.

Table 10-4. Design aspects of libmemkind and libvmemcache

	libmemkind (PMEM)	**libvmemcache**
Allocation Scheme	Dynamic allocator	Extent based (not restricted to sector, page, etc.)
Purpose	General purpose	Lightweight in-memory cache
Fragmentation	Apps with random size allocations/deallocations that run for a longer period	Minimized

libvmemcache Design

libvmemcache has two main design aspects:

1. Allocator design to improve/resolve fragmentation issues
2. A scalable and efficient LRU policy

Extent-based Allocator

libvmemcache can solve fragmentation issues when working with terabyte-sized in-memory workloads and provide high space utilization. Figure 10-5 shows a workload example that creates many small objects, and over time, the allocator stops due to fragmentation.

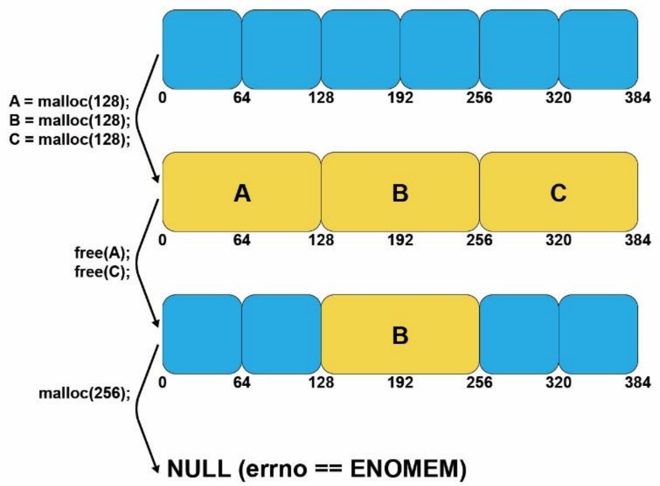

Figure 10-5. An example of a workload that creates many small objects, andthe allocator stops due to fragmentation.

libvmemcache uses an extent-based allocator, where extent is a contiguous set of blocks allocated for storing the data in a database. Extents are typically used with large blocks supported by file systems (sectors, pages, and so on), but such restrictions do not apply when working with persistent memory that supports smaller block sizes (cache-line). Figure 10-6 shows that if a single contiguous free block is not available to allocate an object, multiple, non-contiguous blocks are used to satisfy the allocation request. The non-contiguous allocations appear as a single allocation to the application.

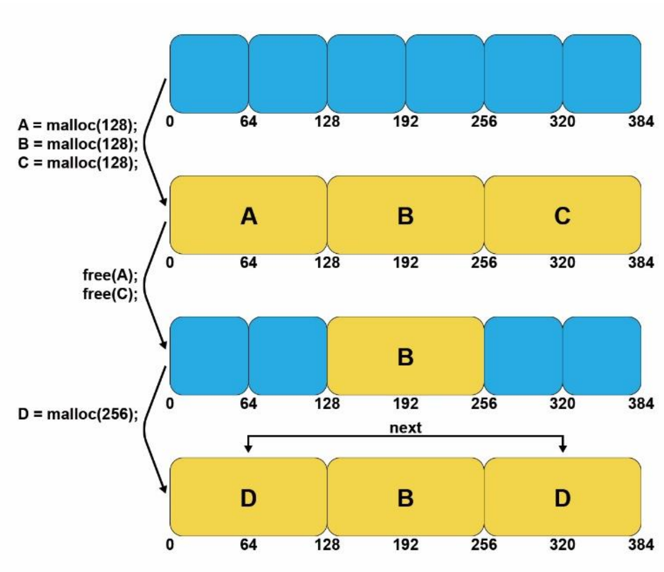

Figure 10-6. Using non-contiguous free blocks to fulfill a larger allocation request.

Scalable Replacement Policy

An LRU cache is traditionally implemented as a doubly-linked list. When an item is retrieved from this list, it gets moved from the middle to the front of the list so it is not evicted. In a multithreaded environment, multiple threads may contend with the front element, all trying to move elements being retrieved to the front element. Therefore, the front element is always locked (along with other locks) before moving the element being retrieved, which results in a few round trips into the kernel. This method is not scalable and is inefficient.

A buffer-based LRU policy creates a scalable and efficient replacement policy. A non-blocking ring buffer is placed in front of the LRU linked list to track the elements being retrieved. When an element is retrieved, it is added to this buffer, and only when the buffer is full (or the element is being evicted), the linked-list is locked and the elements in that buffer are processed and moved to the front of the list. This method preserves the LRU policy and provides a scalable LRU mechanism with minimal performance impact. Figure 10-7 shows a ring buffer-based design for the LRU algorithm.

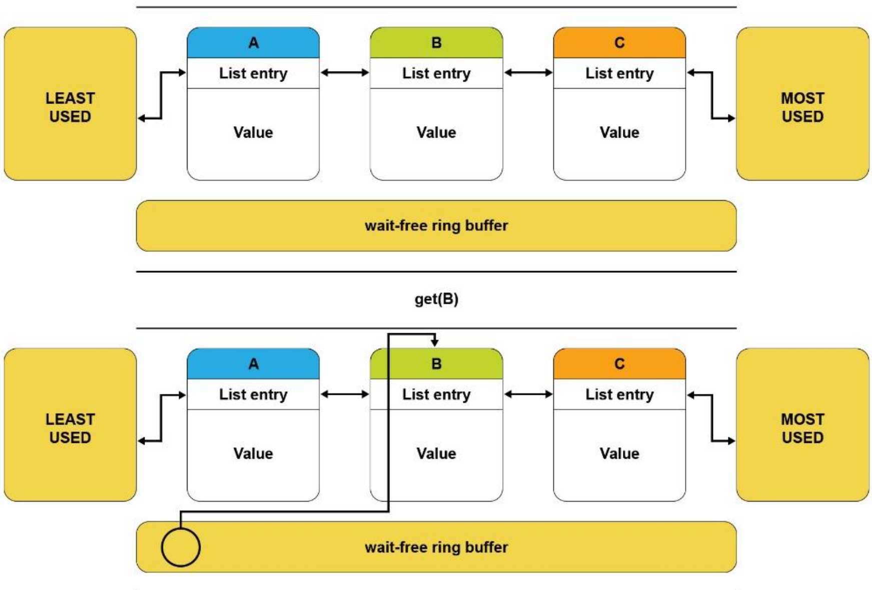

Figure 10-7. A ring buffer-based LRU design.

Using libvmemcache

Table 10-5 lists the basic functions that libvmemcache provides. For a complete list, see the libvmemcache man pages [(https://pmem.io/vmemcache/manpages/master/vmemcache.3.html)](https://pmem.io/vmemcache/manpages/master/vmemcache.3.html).

Table 10-5. The libvmemcache functions.

Function Name	Description
vmemcache_new	Creates an empty unconfigured vmemcache instance with default values: Eviction_policy=VMEMCACHE_REPLACEMENT_LRU Extent_size = VMEMCAHE_MIN_EXTENT VMEMCACHE_MIN_POOL
vmemcache_add	Associates the cache with a path
vmemcache_set_size	Sets the size of the cache
vmemcache_set_extent_size	Sets the block size of the cache (256 bytes minimum)

vmemcache_set_eviction_policy	Sets the eviction policy: 1. VMEMCACHE_REPLACEMENT_NONE 2. VMEMCACHE_REPLACEMENT_LRU
vmemcache_add	Associates the cache with a given path on a DAX-enabled file system or non-DAX enabled file system
vmemcache_delete	Frees any structures associated with the cache
vmemcache_get	Searches for an entry with the given key and if found, the entry's value is copied to `vbuf`
vmemcache_put	Inserts the given `key:value pair` into the cache
vmemcache_evict	Removes the given key from the cache
vmemcache_callback_on_evict	Called when an entry is being removed from the cache
vmemcache_callback_on_miss	Called when a get query fails to provide an opportunity to insert the missing key

To illustrate how libvmemcache is used, Listing 10-12 shows how to create an instance of `vmemcache` using default values. This example uses a temporary file on a DAX-enabled file system and shows how a callback is registered after a cache miss for a key "meow."

Listing 10-12. An example program using libvmemcache.

```
/*
 * vmemcache.c - This example uses a temporary file
 *        on a DAX-enabled file system and
 *        shows how a callback is registered
 *        after a cache miss for a key "meow."
 */

#include <libvmemcache.h>
#include <stdio.h>
#include <string.h>

#define STR_AND_LEN(x) (x), strlen(x)

static VMEMcache *cache;
```

```

static void on_miss(VMEMcache *cache, const void *key,
    size_t key_size, void *arg)
{
    vmemcache_put(cache, STR_AND_LEN("meow"),
         STR_AND_LEN("Cthulhu fthagn"));
}

static void get(const char *key)
{
    char buf[128];
    ssize_t len = vmemcache_get(cache,
    STR_AND_LEN(key), buf, sizeof(buf), 0, NULL);
    if (len >= 0)
        printf("%.*s\n", (int)len, buf);
    else
        printf("(key not found: %s)\n", key);
}

int main()
{
    cache = vmemcache_new();
    if (vmemcache_add(cache, "/pmemfs")) {
        fprintf(stderr, "error: vmemcache_add: %s\n",
                vmemcache_errormsg());
            return 1;
    }

    /* Query a non-existent key. */
    get("meow");

    /* Insert then query. */
    vmemcache_put(cache, STR_AND_LEN("bark"),
        STR_AND_LEN("Lorem ipsum"));
    get("bark");

    /* Install an on-miss handler. */
    vmemcache_callback_on_miss(cache, on_miss, 0);
    get("meow");

    vmemcache_delete(cache);
    return 0;
```

```
89  }
```

- Line 68: Creates a new instance of `vmemcache` with default values for `eviction_policy and extent_size.`
- Line 69: Calls the `vmemcache_add()` function to associate cache with a given path.
- Line 76: Calls the `get()` function to query on an existing key. This function calls the `vmemcache_get()` function with error checking for success/failure of the function.
- Line 79: Calls `vmemcache_put()` to insert a new key.
- Line 84: Adds an on-miss callback handler to insert the key "meow" into the cache.
- Line 85: Retrieves the key "meow" using the `get()` function.
- Line 87: Deletes the `vmemcache` instance.

Summary

This chapter shows how persistent memory's large capacity can be used to hold volatile application data. Applications can choose to allocate and access data from DRAM or persistent memory, or both.

memkind is a very flexible and easy-to-use library with semantics that are similar to the `libc malloc/free` APIs that developers frequently use.

libvmemcache is an embeddable and lightweight in-memory caching solution that allows applications to efficiently use persistent memory's large capacity in a scalable way. libvmemcache is an open-source project available on GitHub at https://github.com/pmem/vmemcache.