CHAPTER 7

libpmemobj: A Native Transactional Object Store

In the previous chapter, we described `libpmem`, the low-level persistent memory library that provides you with an easy way to directly access persistent memory. `libpmem` is a small, lightweight, and feature-limited library that is designed for software that tracks every store to `pmem` and needs to flush those changes to durability. It excels at what it does. However, most developers will find higher-level libraries within the Persistent Memory Development Kit (PMDK), like `libpmemobj`, to be much more convenient.

This chapter describes `libpmemobj`, which builds upon `libpmem` and turns persistent memory-mapped files into a flexible object-store. It supports transactions, memory management, locking, lists, and several other features.

What is libpmemobj?

The `libpmemobj` library provides a transactional object store in persistent memory for applications that require transactions and persistent memory management using direct access (DAX) to the memory. Briefly recapping our DAX discussion in Chapter 3, DAX allows applications to memory-map files on a persistent memory-aware file system to provide direct load/store operations without paging blocks from a block storage device. It bypasses the kernel, avoids context switches and

interrupts, and allows applications to read and write directly to the byte-addressable persistent storage.

Why not malloc()?

Using `libpmem` seems simple. You need to flush anything you have written, and use discipline when ordering such that data needs to be persisted before any pointers to it go live.

If only persistent memory programming were so simple. Apart from some specific patterns that can be done in a simpler way, such as append-only records that can be efficiently handled by `libpmemlog`, any new piece of data needs to have its memory allocated. When and how should the allocator mark the memory as in-use? Should the allocator mark the memory as allocated before writing data or after? Neither approach works for these reasons:

- If the allocator marks the memory as allocated before the data is written, a power outage during the write can cause torn updates and a so-called "persistent leak."
- If the allocator writes the data, then marks it as allocated, a power outage that occurs between the write completing and the allocator marking it as allocated can overwrite the data when the application restarts since the allocator believes the block is available.

Another problem is that a significant number of data structures include cyclical references and thus do not form a tree. They could be implemented as a tree, but this approach is usually harder to implement.

Byte-addressable memory guarantees atomicity of only a single write. For current processors, that is generally one 64-bit word (8-bytes) that should be aligned, but this is not a requirement in practice.

All of the above problems could be solved if multiple writes occurred simultaneously. In the event of a power failure, any incomplete writes should either be replayed as though the power failure never happened or discarded as though the write never occurred. Applications solve this in different ways using atomic operations, transactions, redo/undo logging, etc. Using `libpmemobj` can solve those problems because it uses atomic transactions and redo/undo logs.

Grouping Operations

With the exception of modifying a single scalar value that fits within the processor's word, a series of data modifications must be grouped together and accompanied by a means of detecting an interruption before completion.

Memory Pools

Memory-mapped files are at the core of the persistent memory programming model. The `libpmemobj` library provides a convenient API to easily manage pool creation and access, avoiding the complexity of directly mapping and synchronizing data. PMDK also provides a `pmempool` utility to administer memory pools from the command line. Memory pools reside on DAX-mounted file systems.

Creating Memory Pools

Use the `pmempool` utility to create persistent memory pools for use with applications. Several pool types can be created including `pmemblk`, `pmemlog`, and `pmemobj`. When using `libpmemobj` in applications, you want to create a pool of type `obj (pmemobj)`. Refer to the `pmempool-create(1)` man page for all available commands and options. The examples below are for reference:

Example 1: Create a `libpmemobj (obj)` type pool of minimum allowed size and layout called “my_layout” in the mounted file system /mnt/pmemfs0/

```
$ pmempool create --layout my_layout obj /mnt/pmemfs0/pool.obj
```

Example 2: Create a `libpmemobj (obj)` pool of 20GiB and layout called “my_layout” in the mounted file system /mnt/pmemfs0/

```
$ pmempool create --layout my_layout --size 20G obj \
/mnt/pmemfs0/pool.obj
```

Example 3: Create a `libpmemobj (obj)` pool using all available capacity within the /mnt/pmemfs0/ file system using the layout name of “my_layout”

```
$ pmempool create --layout my_layout --max-size obj \
/mnt/pmemfs0/pool.obj
```

Applications can programmatically create pools using the `pmemobj_create()` that do not exist at application start time. `pmemobj_create()` has the following arguments:

```
PMEMobjpool *pmemobj_create(const char *path,
    const char *layout, size_t poolsize, mode_t mode);
```

- `path` specifies the name of the memory pool file to be created, including a full or relative path to the file.
- `layout` specifies the application’s layout type in the form of a string to identify the pool.
- `poolsize` specifies the required size for the pool. The memory pool file is fully allocated to the size `poolsize` using `posix_fallocate(3)`. The minimum size for a pool is defined as `PMEMOBJ_MIN_POOL` in `<libpmemobj.h>`. If the pool already exists, `pmemobj_create()` will return an `EEXISTS` error. Specifying `poolsize` as zero will take the pool size from the file size and will verify that the file appears to be empty by searching for any non-zero data in the pool header at the beginning of the file.
- `mode` specifies the ACL permissions to use when creating the file, as described by `create(2)`.

Listing 7-1 shows how to create a pool using the `pmemobj_create()` function.

Listing 7-1. pwriter.c – An example showing how to create a pool using pmemobj_create().

```
33  /*
34   * pwriter.c -  Write a string to a
35   *              persistent memory pool
36   */
37
38  #include <stdio.h>
```

```
39  #include <string.h>
40  #include <libpmemobj.h>
41
42  #define LAYOUT_NAME "rweg"
43  #define MAX_BUF_LEN 31
44
45  struct my_root {
46      size_t len;
47      char buf[MAX_BUF_LEN];
48  };
49
50  int
51  main(int argc, char *argv[])
52  {
53      if (argc != 2) {
54          printf("usage: %s file-name\n", argv[0]);
55          return 1;
56      }
57
58      PMEMobjpool *pop = pmemobj_create(argv[1],
59          LAYOUT_NAME, PMEMOBJ_MIN_POOL, 0666);
60
61      if (pop == NULL) {
62          perror("pmemobj_create");
63          return 1;
64      }
65
66      PMEMoid root = pmemobj_root(pop,
67          sizeof(struct my_root));
68
69      struct my_root *rootp = pmemobj_direct(root);
70
71      char buf[MAX_BUF_LEN] = "Hello PMEM World";
72
73      rootp->len = strlen(buf);
74      pmemobj_persist(pop, &rootp->len,
75          sizeof(rootp->len));
76
77      pmemobj_memcpy_persist(pop, rootp->buf, buf,
78          rootp->len);
79
80      pmemobj_close(pop);
```

```
81
82        return 0;
83    }
```

- Line 42: We define the name for our pool layout to be “rweg” (Read-Write Example). This is just a name and can be any string that uniquely identifies the pool to the application. A `NULL` value is valid. In the case where multiple pools are opened by the application, this name uniquely identifies it.
- Line 43: We define the maximum length of the write buffer.
- Lines 45-47: This defines the `root` object data structure which has members `len` and `buf`. `buf` contains the string we want to write and the `len` is the length of the buffer.
- Lines 53- 56: The `pwriter` command accepts one argument: the path and pool name to write to. For example, `/mnt/pmemfs0/helloworld_obj.pool`. The filename extension is arbitrary and optional.
- Lines 58-59: We call `pmemobj_create()` to create the pool using the filename passed in from the command line, the layout name of “rweg,” a size we set to be the minimum size for an object pool type, and permissions of 0666. We cannot create a pool smaller than defined by `PMEMOBJ_MIN_POOL` or larger than the available space on the file system. Since the string in our example is very small, we only require a minimally sized pool. On success, `pmemobj_create()` returns a Pool Object Pointer (POP) of type `PMEMobjpool`, which we can use to acquire a pointer to the `root` object.
- Lines 61-64: If `pmemobj_create()` fails, we will exit the program and return an error.
- Line 66: Using the pop acquired from Line 58, we use the `pmemobj_root()` function to locate the `root` object.
- Line 69: We use the `pmemobj_direct()` function to get a pointer to the root object we found in Line 66.
- Line 71: We set the string/buffer to “Hello PMEM World.”
- Lines 73-78. After determining the length of the buffer, we first write the `len` and then the `buf` member of our `root` object to persistent memory.
- Line 80: We close the persistent memory pool by unmapping it.

Pool Object Pointer (POP) and the Root Object

Due to the address-space layout randomization (ASLR) feature used by most operating systems, the location of the pool—once memory-mapped into the application address space—can differ between executions and system reboots. Without a way to access the data within the pool, you would find it challenging to locate the data within a pool. PMDK-based pools have a small amount of metadata to solve this problem.

Every `pmemobj (obj)` type pool has a `root` object. This `root` object is necessary because it is used as an entry point from which to find all the other objects created in a pool; that is, user data. An application will locate the `root` object using a special object called Pool Object Pointer (POP). The POP object resides in volatile memory and is created with every program invocation. It keeps track of metadata related to the pool, such as the offset to the root object inside the pool. Figure 7-1 depicts the POP and memory pool layout:

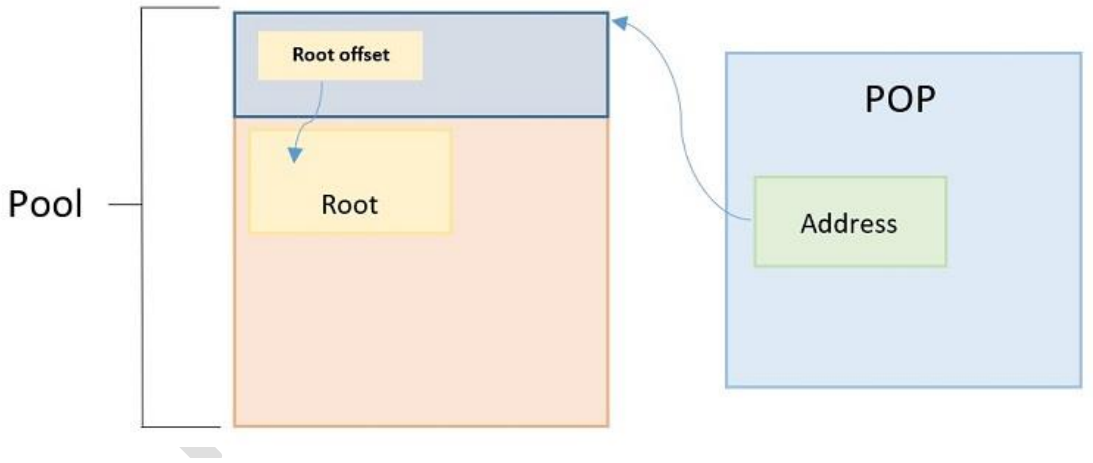

Figure 7-1. A high-level overview of a persistent memory pool with a pool object pointer (POP) pointing to the root object.

Using a valid `pop` pointer, you can use the `pmemobj_root()` function to get a pointer <verb or preposition here?> the `root` object. Internally, this function creates a valid pointer by adding the current memory address of the mapped pool plus the internal offset to the root.

Opening and Reading From Memory Pools

You create a pool using `pmemobj_create()`, and you open an existing pool using `pmemobj_open()`. Both functions return a `PMEMobjpool *pop` pointer. The pwriter example in Listing 7-1 shows how to create a pool and write a string to it. Listing 7-2 shows how to open the same pool to read and display the string.

Listing 7-2. preader.c – An example showing how to open a pool and access the root object and data.

```
33  /*
34   * preader.c -  Read a string from a
35   *              persistent memory pool
36   */
37
38  #include <stdio.h>
39  #include <string.h>
40  #include <libpmemobj.h>
41
42  #define LAYOUT_NAME "rweg"
43  #define MAX_BUF_LEN 31
44
45  struct my_root {
46      size_t len;
47      char buf[MAX_BUF_LEN];
48  };
49
50  int
51  main(int argc, char *argv[])
52  {
53      if (argc != 2) {
54          printf("usage: %s file-name\n", argv[0]);
55          return 1;
56      }
57
58      PMEMobjpool *pop = pmemobj_open(argv[1],
```

```
59          LAYOUT_NAME);
60
61      if (pop == NULL) {
62          perror("pmemobj_open");
63          return 1;
64      }
65
66      PMEMoid root = pmemobj_root(pop,
67          sizeof(struct my_root));
68      struct my_root *rootp = pmemobj_direct(root);
69
70      if (rootp->len == strlen(rootp->buf))
71          printf("%s\n", rootp->buf);
72
73      pmemobj_close(pop);
74
75      return 0;
76  }
```

- Lines 42-48: We use the same data structure declared in pwriter.c. In practice, this should be declared in a header file for consistency.
- Line 58: Open the pool and return a `pop` pointer to it
- Line 66: Upon success, `pmemobj_root()` returns a handle to the `root` object associated with the persistent memory pool `pop`.
- Line 64: `pmemobj_direct()` returns a pointer to the `root` object.
- Lines 70-71: Determine the length of the buffer pointed to by `rootp->buf`. If it matches the length of the buffer we wrote, the contents of the buffer is printed to `STDOUT`.

Memory Poolsets

Individual pools can be resized using the `truncate(1)` command on Linux once an application has stopped and is no longer using it. The capacity of multiple pools can be combined into a *poolset*. Besides providing a way to increase the available space, a poolset can be used to span multiple persistent memory devices and provide both local and remote replication.

You open a poolset the same way as a single pool using `pmemobj_open()`. (At the time of publication, `pmemobj_create()` and the `pmempool` utility cannot create poolsets. Enhancement requests exist for these features.) Although creating poolsets requires manual administration, poolset management can be automated via `libpmempool` or the `pmempool` utility; full details appear in the `poolset(5)` man page.

Concatenated Poolsets

Individual pools can be concatenated using pools on a single or multiple file systems. Concatenation only works with the same pool type: block, object, or log pools. Listing 7-3 shows an example "`myconcatpool.set`" poolset file that concatenates three smaller pools into a larger pool. For illustrative purposes, each pool is a different size and located on different file systems. An application using this poolset would see a single 700GiB memory pool.

Listing 7-3. myconcatpool.set - An example of a concatenated poolset created from three individual pools on three different file systems.

```
PMEMPOOLSET
OPTION NOHDRS
100G /mountpoint0/myfile.part0
200G /mountpoint1/myfile.part1
400G /mountpoint2/myfile.part2
```

Note: Data will be preserved if it exists in `/mountpoint0/myfile.part0`, but any data in `/mountpoint0/myfile.part1` or `/mountpoint0/myfile.part2` will be lost. We recommend that you only add new and empty pools to a poolset.

Replica Poolsets

Besides combining multiple pools to provide more space, a poolset can also maintain multiple copies of the same data to increase resiliency. Data can be replicated to another poolset on a different file of the local host, and a poolset on a remote host.

Listing 7-4 shows a poolset file called "`myreplicatedpool.set`" that will replicate local writes into the `/mnt/pmem0/pool1` pool to another local pool, `/mnt/pmem1/pool1`, on a different file system, and to a `remote-objpool.set` poolset on a remote host called `example.com`.

Listing 7-4. myreplicatedpool.set – An example demonstrating how to replicate local data locally and remote host.

```
PMEMPOOLSET
256G /mnt/pmem0/pool1

REPLICA
256G /mnt/pmem1/pool1

REPLICA user@example.com remote-objpool.set
```

The `librpmem` library, a remote persistent memory support library, underpins this feature. Chapter 18 discusses `librpmem` and replica pools in more detail.

Managing Memory Pools and Poolsets

The `pmempool` utility has several features that developers and system administrators may find useful. We do not present their details here because feature has a detailed man page:

- **pmempool info** prints information and statistics in human-readable format about the specified pool.
- **pmempool check** checks the pool's consistency and repairs pool if it is not consistent.
- **pmempool create** creates a pool of specified type with additional properties specific for this type of pool.
- **pmempool dump** dumps usable data from a pool in hexadecimal or binary format.
- **pmempool rm** removes pool file or all pool files listed in pool set configuration file.

- **pmempool convert** updates the pool to the latest available layout version.
- **pmempool sync** synchronizes replicas within a poolset.
- **pmempool transform** modifies the internal structure of a poolset.
- **pmempool feature** toggle or query a poolset's features.

Typed Object Identifiers (TOIDs)

When we write data to a persistent memory pool or device, we commit it at a physical address. With the ASLR feature of operating systems, when applications open a pool and memory-map it into the address space, the virtual address will change each time. For this reason, a type of handle (pointer) that does not change is needed; this handle is called an OID (Object Identifier). Internally, it is a pair of the pool or poolset unique identifier (UUID) and an offset within the pool or poolset. The OID can be translated back and forth between its persistent form and pointers that are fit for direct use by this particular instance of your program.

At a low level, such translation can be done manually via functions such as `pmemobj_direct()` that appear in the preader.c example in Listing 7-2. Because manual translations require explicit type casts and are error-prone, we recommend tagging every object with a type. This allows some form of type safety, and thanks to macros, can be checked at compile time.

For example, a persistent variable declared via `TOID(struct foo) x` can be read via `D_RO(x)->field`. In a pool with the following layout:

```
POBJ_LAYOUT_BEGIN(cathouse);
POBJ_LAYOUT_TOID(cathouse, struct canaries);
POBJ_LAYOUT_TOID(cathouse, int);
POBJ_LAYOUT_END(cathouse);
```

The field val can be accessed with:

```
TOID(int) val;
TOID_ASSIGN(val, oid_of_val);
D_RW(val) = 42;
return R_RO(val);
```

Allocating Memory

Using `malloc()` to allocate memory is quite normal to C developers and those who use languages that do not fully handle automatic memory allocation and deallocation. For persistent memory, you can use `pmemobj_alloc()`, `pmemobj_reserve()`, or `pmemobj_xreserve()` to reserve memory for a transient object and use it the same way you would use `malloc()`. We recommend that you free allocated memory using `pmemobj_free()` or `POBJ_FREE()` when the application no longer requires it to avoid a runtime memory leak. Because these are volatile memory allocations, they will not cause a persistent leak after a crash or graceful application exit.

Persisting Data

The typical intent of using persistent memory is to save data persistently. For this, you need to use one of three APIs that `libpmemobj` provides:

- Atomic operations
- Reserve/Publish
- Transactional

Atomic Operations

The `pmemobj_alloc()` and its variants shown below are easy to use, but they are limited in features so additional coding is required by the developer:

```
int pmemobj_alloc(PMEMobjpool *pop, PMEMoid *oidp,
    size_t size, uint64_t type_num, pmemobj_constr
    constructor, void *arg);
int pmemobj_zalloc(PMEMobjpool *pop, PMEMoid *oidp,
    size_t size, uint64_t type_num);
void pmemobj_free(PMEMoid *oidp);
int pmemobj_realloc(PMEMobjpool *pop, PMEMoid *oidp,
    size_t size, uint64_t type_num);
```

```
int pmemobj_zrealloc(PMEMobjpool *pop, PMEMoid *oidp,
    size_t size, uint64_t type_num);
int pmemobj_strdup(PMEMobjpool *pop, PMEMoid *oidp,
    const char *s, uint64_t type_num);
int pmemobj_wcsdup(PMEMobjpool *pop, PMEMoid *oidp,
    const wchar_t *s, uint64_t type_num);
```

The TOID-based wrappers for most of these functions include:

```
POBJ_NEW(PMEMobjpool *pop, TOID *oidp, TYPE,
    pmemobj_constr constructor, void *arg)
POBJ_ALLOC(PMEMobjpool *pop, TOID *oidp, TYPE, size_t size,
    pmemobj_constr constructor, void *arg)
POBJ_ZNEW(PMEMobjpool *pop, TOID *oidp, TYPE)
POBJ_ZALLOC(PMEMobjpool *pop, TOID *oidp, TYPE, size_t size)
POBJ_REALLOC(PMEMobjpool *pop, TOID *oidp, TYPE, size_t size)
POBJ_ZREALLOC(PMEMobjpool *pop, TOID *oidp, TYPE, size_t size)
POBJ_FREE(TOID *oidp)
```

These functions reserve the object in a temporary state, call the constructor you provided, and then in one atomic action, marks the allocation as persistent. They will insert the pointer to the newly initialized object into a variable of your choice.

If the new object needs to be merely zeroed, `pmemobj_zalloc()` does so without requiring a constructor.

Because copying NULL-terminated strings is a common operation, `libpmemobj` provides `pmemobj_strdup()` and its wide-char variant `pmemobj_wcsdup()` to handle this. `pmemobj_strdup()` provides the same semantics as `strdup(3)`, but operates on the persistent memory heap associated with the memory pool.

Once you are done with the object, `pmemobj_free()` will deallocate the object while zeroing the variable that stored the pointer to it. The `pmemobj_free()` function frees the memory space represented by `oidp`, which must have been allocated by a previous call to `pmemobj_alloc()`, `pmemobj_xalloc()`, `pmemobj_zalloc()`, `pmemobj_realloc()`, or `pmemobj_zrealloc()`. The `pmemobj_free()` function provides the same semantics as `free(3)`, but instead of operating on the process heap supplied by the system, it operates on the persistent memory heap.

Listing 7-5 shows a small example of allocating and freeing memory using the `libpmemobj` API.

Listing 7-5. Using pmemobj_alloc() to allocate memory and using pmemobj_free() to free it.

```
33  /*
34   * pmemobj_alloc.c - An example to show how to use
35   *                   pmemobj_alloc()
36   */
..
47  typedef uint32_t color;
48
49  static int paintball_init(PMEMobjpool *pop,
50          void *ptr, void *arg)
51  {
52      *(color *)ptr = time(0) & 0xffffff;
53      pmemobj_persist(pop, ptr, sizeof(color));
54      return 0;
55  }
56
57  int main()
58  {
59      PMEMobjpool *pool = pmemobj_open(POOL, LAYOUT);
60      if (!pool) {
61          pool = pmemobj_create(POOL, LAYOUT,
62          PMEMOBJ_MIN_POOL, 0666);
63          if (!pool)
64              die("Couldn't open pool: %m\n");
65
66      }
67      PMEMoid root = pmemobj_root(pool,
68              sizeof(PMEMoid) * 6);
69      if (OID_IS_NULL(root))
70          die("Couldn't access root object.\n");
71
72      PMEMoid *chamber = (PMEMoid *)pmemobj_direct(root)
73          + (getpid() % 6);
74      if (OID_IS_NULL(*chamber)) {
75          printf("Reloading.\n");
76          if (pmemobj_alloc(pool, chamber, sizeof(color)
77              , 0, paintball_init, 0))
```

```
78              die("Failed to alloc: %m\n");
79      } else {
80          printf("Shooting %06x colored bullet.\n",
81          *(color *)pmemobj_direct(*chamber));
82          pmemobj_free(chamber);
83      }
84
85      pmemobj_close(pool);
86      return 0;
87  }
```

- Line 47: Defines a color that will be stored in the pool.
- Lines 49-54: The `paintball_init()` function is called when we allocate memory (Line 76). This function takes a pool and object pointer, calculates a random hex value for the paintball color, and persistently writes it to the pool. The program exits when the write completes.
- Lines 59-70: Opens or creates a pool and acquires a pointer to the root object within the pool.
- Line 70: Obtain a pointer to an offset within the pool.
- Lines 74-78: If the pointer in line 70 is not a valid object, we allocate some space and call `paintball_init()`.
- Lines 79-80: If the pointer in line 70 is a valid object, we read the color value, print the string, and free the object.

Reserve/Publish API

The atomic allocation API will not help if:

- There is more than one reference to the object that needs to be updated.
- There are multiple scalars that need to be updated.

For example, if your program needs to subtract money from account A and add it to account B, both operations must be done together. This can be done via the reserve/publish API.

To use it, you specify any number of operations to be done. The operations may be setting a scalar 64-bit value using `pmemobj_set_value()`, freeing an object with `pmemobj_defer_free()`, or allocating it using `pmemobj_reserve()`. Of these, only the allocation happens immediately, letting you do any initialization of

the newly reserved object. Modifications will not become persistent until `pmemobj_publish()` is called.

Functions provided by `libpmemobj` related to the reserve/publish feature are:

```
PMEMoid pmemobj_reserve(PMEMobjpool *pop,
    struct pobj_action *act, size_t size, uint64_t type_num);
void pmemobj_defer_free(PMEMobjpool *pop, PMEMoid oid,
    struct pobj_action *act);
void pmemobj_set_value(PMEMobjpool *pop,
    struct pobj_action *act, uint64_t *ptr, uint64_t value);
int pmemobj_publish(PMEMobjpool *pop,
    struct pobj_action *actv, size_t actvcnt);
void pmemobj_cancel(PMEMobjpool *pop,
    struct pobj_action *actv, size_t actvcnt);
```

Listing 7-6 is a simple banking example that demonstrates how to change multiple scalars (account balances) before publishing the updates into the pool.

Listing 7-6. Using the reserve/publish API to modify bank account balances.

```
32
33  /*
34   * reserve_publish.c – An example using the
35   *                     reserve/publish libpmemobj API
36   */
37
..
44  #define POOL "/mnt/pmem/balance"
45
46  static PMEMobjpool *pool;
47
48  struct account {
49      PMEMoid name;
50      uint64_t balance;
51  };
52  TOID_DECLARE(struct account, 0);
53
..
60  static PMEMoid new_account(const char *name,
61                  int deposit)
```

```
62  {
63      int len = strlen(name) + 1;
64
65      struct pobj_action act[2];
66      PMEMoid str = pmemobj_reserve(pool, act + 0,
67                      len, 0);
68      if (OID_IS_NULL(str))
69          die("Can't allocate string: %m\n");
..
75      pmemobj_memcpy(pool, pmemobj_direct(str), name,
76                      len, PMEMOBJ_F_MEM_NODRAIN);
77      TOID(struct account) acc;
78      PMEMoid acc_oid = pmemobj_reserve(pool, act + 1,
79                      sizeof(struct account), 1);
80      TOID_ASSIGN(acc, acc_oid);
81      if (TOID_IS_NULL(acc))
82          die("Can't allocate account: %m\n");
83      D_RW(acc)->name = str;
84      D_RW(acc)->balance = deposit;
85      pmemobj_persist(pool, D_RW(acc),
86                      sizeof(struct account));
87      pmemobj_publish(pool, act, 2);
88      return acc_oid;
89  }
90
91  int main()
92  {
93      if (!(pool = pmemobj_create(POOL, "",
94                              PMEMOBJ_MIN_POOL, 0600)))
95          die("Can't create pool "%s": %m\n", POOL);
96
97      TOID(struct account) account_a, account_b;
98      TOID_ASSIGN(account_a,
99                      new_account("Julius Caesar", 100));
100     TOID_ASSIGN(account_b,
101                     new_account("Mark Anthony", 50));
102
103     int price = 42;
104     struct pobj_action act[2];
105     pmemobj_set_value(pool, &act[0],
106                     &D_RW(account_a)->balance,
107                     D_RW(account_a)->balance - price);
```

```
108      pmemobj_set_value(pool, &act[1],
109                        &D_RW(account_b)->balance,
110                        D_RW(account_b)->balance + price);
111      pmemobj_publish(pool, act, 2);
112
113      pmemobj_close(pool);
114      return 0;
115  }
```

- Line 44: Defines the location of the memory pool.
- Lines 48-52: Declares an account data structure with a `name` and `balance`.
- Lines 60-89: The `new_account()` function reserves the memory (Lines 66 and 78), updates the name and balance (Lines 83 and 84), persists the changes (Line 85), and then publishes the updates (Line 87).
- Lines 93-95: Create a new pool or exit on failure.
- Line 97: Delcare two account instances.
- Lines 98-101: Create a new account for each owner with initial balances.
- Lines 103-111: We subtract 42 from Julius Caesar's account, and add 42 to Mark Anthony's account. The modifications are published on Line 111.

Transactional API

The reserve/publish API is fast, but it does not allow reading data you have just written. In such cases, you can use the transactional API.

The first time a variable is written, it must be explicitly added to the transaction. This can be done via `pmemobj_tx_add_range()` or its variants (xadd, `_direct`). Convenient macros such as `TX_ADD()` or `TX_SET()` can perform the same operation. The transaction-based functions and macros provided by `libpmemobj` include:

```
int pmemobj_tx_add_range(PMEMoid oid, uint64_t off,
    size_t size);
```

```
int pmemobj_tx_add_range_direct(const void *ptr, size_t size);

TX_ADD(TOID o)
TX_ADD_FIELD(TOID o, FIELD)
TX_ADD_DIRECT(TYPE *p)
TX_ADD_FIELD_DIRECT(TYPE *p, FIELD)

TX_SET(TOID o, FIELD, VALUE)
TX_SET_DIRECT(TYPE *p, FIELD, VALUE)
TX_MEMCPY(void *dest, const void *src, size_t num)
TX_MEMSET(void *dest, int c, size_t num)
```

The transaction may also allocate entirely new objects, reserve their memory, and then persistently allocate them only one transaction commit. These functions include:

```
PMEMoid pmemobj_tx_alloc(size_t size, uint64_t type_num);
PMEMoid pmemobj_tx_zalloc(size_t size, uint64_t type_num);
PMEMoid pmemobj_tx_realloc(PMEMoid oid, size_t size,
    uint64_t type_num);
PMEMoid pmemobj_tx_zrealloc(PMEMoid oid, size_t size,
    uint64_t type_num);
PMEMoid pmemobj_tx_strdup(const char *s, uint64_t type_num);
PMEMoid pmemobj_tx_wcsdup(const wchar_t *s,
    uint64_t type_num);
```

We can re-write the banking example from Listing 7-6 using the transaction API. Most of the code remains the same except when we want to add or subtract amounts from the balance; we encapsulate those updates in a transation, as shown in Listing 7-7.

Listing 7-7. Using the transaction API to modify bank account balances.

```
33  /*
34   * tx.c - An example using the transaction API
35   */
36
..
94  int main()
95  {
```

```
 96      if (!(pool = pmemobj_create(POOL, "",
 97                          PMEMOBJ_MIN_POOL, 0600)))
 98          die("Can't create pool "%s": %m\n", POOL);
 99
100      TOID(struct account) account_a, account_b;
101      TOID_ASSIGN(account_a,
102                  new_account("Julius Caesar", 100));
103      TOID_ASSIGN(account_b,
104                  new_account("Mark Anthony", 50));
105
106      int price = 42;
107      TX_BEGIN(pool) {
108          TX_ADD_DIRECT(&D_RW(account_a)->balance);
109          TX_ADD_DIRECT(&D_RW(account_b)->balance);
110          D_RW(account_a)->balance -= price;
111          D_RW(account_b)->balance += price;
112      } TX_END
113
114      pmemobj_close(pool);
115      return 0;
116  }
```

- Line 107: We start the transaction.
- Lines 108-111: Make balance modifications to multiple accounts.
- Line 112: Finish the transaction. All updates will either complete entirely or they will be rolled back if the application or system crashes before the transaction completes.

Each transaction has multiple stages in which an application can interact. These transaction stages include:

- `TX_STAGE_NONE`: No open transaction in this thread.
- `TX_STAGE_WORK`: Transaction in progress.
- `TX_STAGE_ONCOMMIT`: Successfully committed.
- `TX_STAGE_ONABORT`: The transaction start either failed or was aborted.
- `TX_STAGE_FINALLY`: Ready for clean up.

The example in Listing 7-7 uses the two mandatory stages: `TX_BEGIN` and `TX_END`. However, we could easily have added the other stages to perform actions for the other stages; for example:

```
TX_BEGIN(Pop) {
      /* the actual transaction code goes here... */
} TX_ONCOMMIT {
      /*
       * optional - executed only if the above block
       * successfully completes
       */
} TX_ONABORT {
      /*
       * optional - executed only if starting the transaction
 * fails, or if transaction is aborted by an error or a
 * call to pmemobj_tx_abort()
       */
} TX_FINALLY {
      /*
       * optional - if exists, it is executed after
       * TX_ONCOMMIT or TX_ONABORT block
       */
} TX_END /* mandatory */
```

Optionally, you can provide a list of parameters for the transaction. Each parameter consists of a type followed by one of these type-specific number of values:

- `TX_PARAM_NONE` is used as a termination marker with no following value.
- `TX_PARAM_MUTEX` is followed by one value, a pmem-resident `PMEMmutex`.
- `TX_PARAM_RWLOCK` is followed by one value, a pmem-resident `PMEMrwlock`.
- `TX_PARAM_CB` is followed by two values: a callback function of type `pmemobj_tx_callback` and a void pointer.

Using `TX_PARAM_MUTEX` or `TX_PARAM_RWLOCK` causes the specified lock to be acquired at the beginning of the transaction. `TX_PARAM_RWLOCK` acquires the lock for writing. It is guaranteed that `pmemobj_tx_begin()` will acquire all locks

prior to successful completion, and they will be held by the current thread until the outermost transaction is finished. Locks are taken in order from left to right. To avoid deadlocks, you are responsible for proper lock ordering.

`TX_PARAM_CB` registers the specified callback function to be executed at each transaction stage. `For TX_STAGE_WORK`, the callback is executed prior to commit. For all other stages, the callback is executed as the first operation after a stage change. It will also be called after each transaction; in this case the stage parameter will be set to <what?>,

Optional Flags

Many of the functions discussed for the atomic, reserve/publish, and transactional APIs have a variant with a "flags" argument that accepts these values :

- `POBJ_XALLOC_ZERO` zeroes the object allocated.
- `POBJ_XALLOC_NO_FLUSH` suppresses automatic flushing. It is expected that you flush the data in some way; otherwise, it may not be durable in case of an unexpected power loss.

Persisting Data Summary

The atomic, reserve/publish, and transactional APIs have different strengths:

- Atomic allocations are the simplest and fastest, but their use is limited to allocating and initializing wholly new blocks.
- The reserve/publish API can be as fast as atomic allocations when all operations involve either allocating or deallocating whole objects, or modifying scalar values. However, being able to read the data you have just written may be desirable.
- The transactional API requires slow synchronization whenever a variable is added to the transaction. If the variable is changed multiple times during the transaction, subsequent operations are free. It also allows conveniently mutating pieces of data larger than a single machine word.

ACID Properties

ACID (Atomicity, Consistency, Isolation, Durability) is a set of transaction properties intended to guarantee validity even in the event of errors, power failures, etc.

ACID transactions are commonly found in databases where a sequence of database operations that satisfies the ACID properties is called a *transaction*. The transaction can be perceived as a single logical operation on the data. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction. We use the same approach for persistent memory programming when we want to update one or more objects, a variable, a data structure, or a member of a data structure.

In the context of databases, we describe each property as follows:

- **Atomicity.** This property states that a transaction must be treated as an atomic unit; that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency.** The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must also remain consistent after the transaction is executed.
- **Isolation.** In a database system where more than one transaction is being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.
- **Durability.** The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written onto the disk, then that data will be updated once the system goes back into action.

The distinct parallels with persistent memory programming for which we can describe each ACID property are as follows:

- **Atomicity.** A transaction is (fail-safe) atomic if it can happen either in its entirety or not at all. At no point interrupting it, by a power loss, crashing or exiting the program, or canceling the transaction, may leave it partially done.
- **Consistency.** The data structures (either of the allocator or of your program) remain in a valid state, even in an unexpected crash. `libpmemobj` will ensure consistency of its own state and, as long as all changes to your data structures are guarded with one of the three provided APIs, will also help you attain consistency of that data.
- **Isolation.** This is important when multiple threads access the same database/memory pool/object store. `libpmemobj` provides full isolation only for its own structures.
- **Durability.** Once a transaction has been committed, a power loss cannnot possibly cause your writes to be lost—barring, of course, failure of the hardware itself.

Guarantees of libpmemobj's APIs

The transactional, atomic allocation, and reserve/publish APIs within `libpmemobj` all provide fail-safe atomicity and consistency.

The transactional API ensures the durability of any modifications of memory for an object that has been added to the transaction. An exception is when the `POBJ_X***_NO_FLUSH` flag is used, in which case the application is responsible for either flushing that memory range itself or using the `memcpy-like` functions from `libpmemobj`. The no-flush flag does not provide any isolation between threads, meaning partial writes are immediately visible to other threads.

The atomic allocation API requires that applications flush the writes done by the object's constructor. This ensures durability if the operation succeeded. It is the only API that provides full isolation between threads.

The reserve/publish API requires explicit flushes of writes to memory blocks allocated via `pmemobj_reserve()` that will flush writes done via `pmemobj_set_value()`. There is no isolation between threads, although no

modifications go live until `pmemobj_publish()` starts, allowing you to take explicit locks for just the publishing stage.

Using terms known from databases, the isolation levels provided are:

- Transactional API: `READ_UNCOMMITTED`
- Atomic allocations API: `READ_COMMITTED`
- Reserve/Publish API: `READ_COMMITTED` until publishing starts, then `READ_UNCOMMITTED`

Managing Library Behavior

The `pmemobj_set_funcs()` function allows an application to override memory allocation calls used internally by `libpmemobj`. Passing in `NULL` for any of the handlers will cause the `libpmemobj` default function to be used. The library does not make heavy use of the system `malloc()` functions, but it does allocate approximately 4-8 kilobytes for each memory pool in use.

By default, `libpmemobj` supports up to 1024 parallel transactions/allocations. For debugging purposes, it is possible to decrease this value by setting the `PMEMOBJ_NLANES` shell environment variable to the desired limit. For example at the shell prompt, run "`export PMEMOBJ_NLANES=512`" then run the application:

```
$ export PMEMOBJ_NLANES=512
$ ./my_app
```

To return to the default behavior, unset `PMEMOBJ_NLANES` using:

```
$ unset PMEMOBJ_NLANES
```

Debugging and Error Handling

If an error is detected during the call to a `libpmemobj` function, the application may retrieve an error message describing the reason for the failure from `pmemobj_errormsg()`. This function returns a pointer to a static buffer containing the last error message logged for the current thread. If `errno` was set, the error

message may include a description of the corresponding error code as returned by `strerror(3)`. The error message buffer is thread-local; errors encountered in one thread do not affect its value in other threads. The buffer is never cleared by any library function; its content is significant only when the return value of the immediately preceding call to a `libpmemobj` function indicated an error, or if `errno` was set. The application must not modify or free the error message string, but it may be modified by subsequent calls to other library functions.

Two versions of `libpmemobj` are typically available on a development system. The non-debug version is optimized for performance and used when a program is linked using the `-lpmemobj` option. This library skips checks that impact performance, never logs any trace information, and does not perform any run-time assertions.

A debug version of `libpmemobj` is provided and available in `/usr/lib/pmdk_debug` or `/usr/local/lib64/pmdk_debug`. The debug version contains run-time assertions and tracepoints.

The common way to use the debug version is to set the environment variable `LD_LIBRARY_PATH`. Alternatively, you can use `LD_PRELOAD` to point to `/usr/lib/pmdk_debug` or `/usr/lib64/pmdk_debug`, as appropriate. These libraries may reside in a different location, such as `/usr/local/lib/pmdk_debug` and `/usr/local/lib64/pmdk_debug`, depending on your Linux distribution or if you compiled installed PMDK from source code and chose `/usr/local` as the installation path. The following examples are equivalent methods for loading and using the debug versions of `libpmemobj` with an application called `my_app`:

```
$ export LD_LIBRARY_PATH=/usr/lib64/pmdk_debug
$ ./my_app

Or

$ LD_PRELOAD=/usr/lib64/pmdk_debug ./my_app
```

The output provided by the debug library is controlled using the `PMEMOBJ_LOG_LEVEL` and `PMEMOBJ_LOG_FILE` environment variables. These variables have no effect on the non-debug version of the library.

PMEMOBJ_LOG_LEVEL

The value of `PMEMOBJ_LOG_LEVEL` enables tracepoints in the debug version of the library, as follows:

1. This is the default level when `PMEMOBJ_LOG_LEVEL` is not set. No log messages are emitted at this level.
2. Additional details on any errors detected are logged, in addition to returning the errno-based errors as usual. The same information may be retrieved using `pmemobj_errormsg()`.
3. A trace of basic operations is logged.
4. Enables an extensive amount of function-call tracing in the library.
5. Enables voluminous and fairly obscure tracing information that is likely only useful to the `libpmemobj` developers.

Debug output is written to `STDERR` unless `PMEMOBJ_LOG_FILE` is set. To set a debug level, use:

```
$ export PMEMOBJ_LOG_LEVEL=2
$ ./my_app
```

PMEMOBJ_LOG_FILE

The value of `PMEMOBJ_LOG_FILE` includes the full path and filename of a file where all logging information should be written. If `PMEMOBJ_LOG_FILE` is not set, logging output is written to `STDERR`.

The following example defines the location of the log file to `/var/tmp/libpmemobj_debug.log`, ensures we are using the debug version of `libpmemobj` when executing `my_app` in the background, sets the debug log level to 2, and monitors the log in real-time using `tail -f`:

```
$ export PMEMOBJ_LOG_FILE=/var/tmp/libpmemobj_debug.log
$ export PMEMOBJ_LOG_LEVEL=2
$ LD_PRELOAD=/usr/lib64/pmdk_debug ./my_app &
$ tail -f /var/tmp/libpmemobj_debug.log
```

If the last character in the debug log filename is "`-`", the process identifier (PID) of the current process will be appended to the filename when the log file is created. This is useful if you are debugging multiple processes.

Summary

This chapter describes the `libpmemobj` library, which is designed to simplify persistent memory programming. By providing APIs that deliver atomic operations, transactions, and reserve/publish features, it makes creating applications less error-prone while delivering guarantees for data integrity.