CHAPTER 6

libpmem: Low-Level Persistent Memory Support

This chapter introduces `libpmem`, one of the smallest libraries in PMDK. This C library is very low-level, dealing with things like CPU instructions related to persistent memory, optimal ways to copy data to persistence, and file mapping. Programmers who only want completely raw access to persistent memory, without libraries to provide allocators or transactions, will likely want to use `libpmem` as a basis for their development.

The code in `libpmem` that detects the available CPU instructions, for example, is a mundane boilerplate code that you do not want to invent repeatedly in applications. Leveraging this small amount of code from `libpmem` will save time, and you get the benefit of fully tested and tuned code in the library.

For most programmers, `libpmem` is too low-level, and you can safely skim this chapter quickly (or skip it altogether) and move on to the higher-level, friendlier libraries available in PMDK. All the PMDK libraries that deal with persistence, such as `libpmemobj`, are built on top of `libpmem` to meet their low-level needs.

Like all PMDK libraries, online man pages are available. For `libpmem`, they are at http://pmem.io/pmdk/libpmem/. This site includes links to the man pages for both the Linux and Windows version. Although the goal of the PMDK project was to make the interfaces similar across operating systems, some small differences appear as necessary. The C code examples used in this chapter build and run on both Linux and Windows.

The examples used in this chapter are:

- `simple_copy.c` is a small program that copies a 4k block from a source file to a destination file on persistent memory.
- `full_copy.c` is a more complete copy program, copying the entire file.
- `manpage.c` is the simple example used in the `libpmem` man page.

Using the Library

To use `libpmem`, start by including the appropriate header, as shown in Listing 6-1.

Listing 6-1. Including the libpmem headers.

```
32
33  /*
34   * simple_copy.c
35   *
36   * usage: simple_copy src-file dst-file
37   *
38   * Reads 4k from src-file and writes it to dst-file.
39   */
40
41  #include <sys/types.h>
42  #include <sys/stat.h>
43  #include <fcntl.h>
44  #include <stdio.h>
45  #include <errno.h>
46  #include <stdlib.h>
47  #ifndef _WIN32
48  #include <unistd.h>
49  #else
50  #include <io.h>
51  #endif
52  #include <string.h>
53  #include <libpmem.h>
```

Notice the `include` on line 53. To use `libpmem`, use this include line, and link the C program with `libpmem` using the `-lpmem` option when building under Linux.

Mapping a File

The `libpmem` library contains some convenience functions for memory mapping files. Of course, your application can call `mmap()` on Linux or `MapViewOfFile()` on Windows directly, but using `libpmem` has some advantages:

- `libpmem` knows the correct arguments to the operating system mapping calls. For example, on Linux, it is not safe to flush changes to persistent memory using the CPU instructions directly unless the mapping is created with the `MAP_SYNC` flag to `mmap()`.
- `libpmem` detects if the mapping is actually persistent memory and if using the CPU instructions directly for flushing is safe.

Listing 6-2 shows how to memory map a file on a persistent memory-aware file system into the application.

Listing 6-2. Mapping a persistent memory file.

```
80      /* create a pmem file and memory map it */
81      if ((pmemaddr = pmem_map_file(argv[2], BUF_LEN,
82              PMEM_FILE_CREATE|PMEM_FILE_EXCL,
83              0666, &mapped_len, &is_pmem)) == NULL) {
84          perror("pmem_map_file");
85          exit(1);
86      }
```

As part of the persistent memory detection mentioned above, the flag `is_pmem` is returned by `pmem_map_file`. It is the caller's responsibility to use this flag to determine how to flush changes to persistence. When making a range of memory persistent, the caller can use the optimal flush provided by `libpmem`, `pmem_persist`, only if the `is_pmem` flag is set. This is illustrated in the man page example excerpt in Listing 6-3:

Listing 6-3. manpage.c: Using the is_pmem flag.

```
74      /* Flush above strcpy to persistence */
75      if (is_pmem)
76          pmem_persist(pmemaddr, mapped_len);
77      else
78          pmem_msync(pmemaddr, mapped_len);
```

Listing 6-3 shows the convenience function `pmem_msync()`, which is just a small wrapper around `msync`() or the Windows equivalent. You do not need to build in different logic for Linux and Windows because `libpmem` handles this.

Copying to Persistent Memory

There are several interfaces in `libpmem` for optimally copying or zeroing ranges of persistent memory. The simplest interface shown in Listing 6-4 is used to copy the block of data from the source file to the persistent memory in the destination file, and flush it to persistence.

Listing 6-4. simple_copy.c: Copying to persistent memory.

```
88      /* read up to BUF_LEN from srcfd */
89      if ((cc = read(srcfd, buf, BUF_LEN)) < 0) {
90          pmem_unmap(pmemaddr, mapped_len);
91          perror("read");
92          exit(1);
93      }
94
95      /* write it to the pmem */
96      if (is_pmem) {
97          pmem_memcpy_persist(pmemaddr, buf, cc);
98      } else {
99          memcpy(pmemaddr, buf, cc);
100         pmem_msync(pmemaddr, cc);
101     }
```

Notice how the `is_pmem` flag on line 96 is used just like it would be for calls to `pmem_persist()`, since the `pmem_memcpy_persist()` function includes the flush to persistence.

The interface `pmem_memcpy_persist()` includes the flush to persistent because it may determine the copy is more optimally performed by using non-temporal stores, which bypass the CPU cache and do not require subsequent cache flush instructions for persistence. By providing this API, which both copies and flushes, `libpmem` is free to use the most optimal way to perform both steps.

Separating the Flush Steps

Flushing to persistence involves two steps:

1. Flush the CPU caches or bypass them entirely as explained in the previous example.
2. Wait for any hardware buffers to drain, to ensure writes have reached the media.

These steps are performed together when `pmem_persist()` is called, or they can be called individually by calling `pmem_flush()` for the first step and `pmem_drain()` for the second. Note that either of these steps may be unnecessary on a given platform, and the library knows how to check for that and do what is correct. For example, on Intel® platforms, `pmem_drain` is an empty function.

When does it make sense to break flushing into steps? The example in Listing 6-5 illustrates one reason you might want to do this. Since the example copies data using multiple calls to `memcpy()`, it uses the version of `libpmem` copy (`pmem_memcpy_nodrain()`) that only performs the flush, postponing the final drain step to the end. This works because, unlike the flush step, the drain step does not take an address range; it is a system-wide drain operation so can happen at the end of the loop that copies individual blocks of data.

Listing 6-5. full_copy.c: Separating the flush steps.

```
58  /*
59   * do_copy_to_pmem
60   */
61  static void
62  do_copy_to_pmem(char *pmemaddr, int srcfd, off_t len)
63  {
64      char buf[BUF_LEN];
65      int cc;
66
```

```
67      /*
68       * Copy the file,
69       * saving the last flush & drain step to the end
70       */
71      while ((cc = read(srcfd, buf, BUF_LEN)) > 0) {
72          pmem_memcpy_nodrain(pmemaddr, buf, cc);
73          pmemaddr += cc;
74      }
75
76      if (cc < 0) {
77          perror("read");
78          exit(1);
79      }
80
81      /* Perform final flush step */
82      pmem_drain();
83  }
```

In Listing 6-5, `pmem_memcpy_nodrain()` is specifically designed for persistent memory. When using other libraries and standard functions like `memcpy()`, remember they were written before persistent memory existed and do not perform any flushing to persistence. In particular, the `memcpy()` provided by the C run-time environment often chooses between regular stores (which require flushing) and non-temporal stores (which do not require flushing). It is making that choice based on performance, not persistence. Since you will not know which instructions it chooses, you will need to perform the flush to persistence yourself using `pmem_persist()` or `msync()`.

The choice of instructions used when copying ranges to persistent memory is fairly important to the performance in many applications. The same is true when zeroing out ranges of persistent memory. To meet these needs, `libpmem` provides `pmem_memmove()`, `pmem_memcpy()`, and `pmem_memset()`, which all take a *flags* argument to give the caller more control over which instructions they use. For example, passing the flag `PMEM_F_MEM_NONTEMPORAL` will tell these functions to use non-temporal stores instead of choosing which instructions to use based on the size of the range. The full list of flags is documented in the man pages for these functions.

Summary

This chapter demonstrated some of the fairly small set of APIs provided by `libpmem`. This library does not track what changed for you, does not provide powerfail-safe transactions, and does not provide an allocator. Libraries like `libpmemobj` (described in the next chapter) provide all those tasks and use `libpmem` internally for simple flushing and copying.