CHAPTER 3

Operating System Support for Persistent Memory

In the previous chapter, we described how persistent memory is incorporated into the computer hardware architecture. In this chapter, we describe how operating systems manage persistent memory as a platform resource and describe the options they provide for applications to use persistent memory. We start by comparing memory and storage in popular computer architectures and then describe how operating systems have been extended for persistent memory.

Operating System Support for Memory and Storage

Figure 3-1 shows a simplified view of how operating systems manage storage and volatile memory. As shown, the volatile main memory is attached directly to the CPU through a memory bus. The operating system manages the mapping of memory regions directly into the application's visible memory address space. Storage, which usually operates at speeds much slower than the CPU, is attached through an input/output (I/O) controller. The operating system handles access to the storage through device driver modules loaded into the operating systems I/O subsystem.

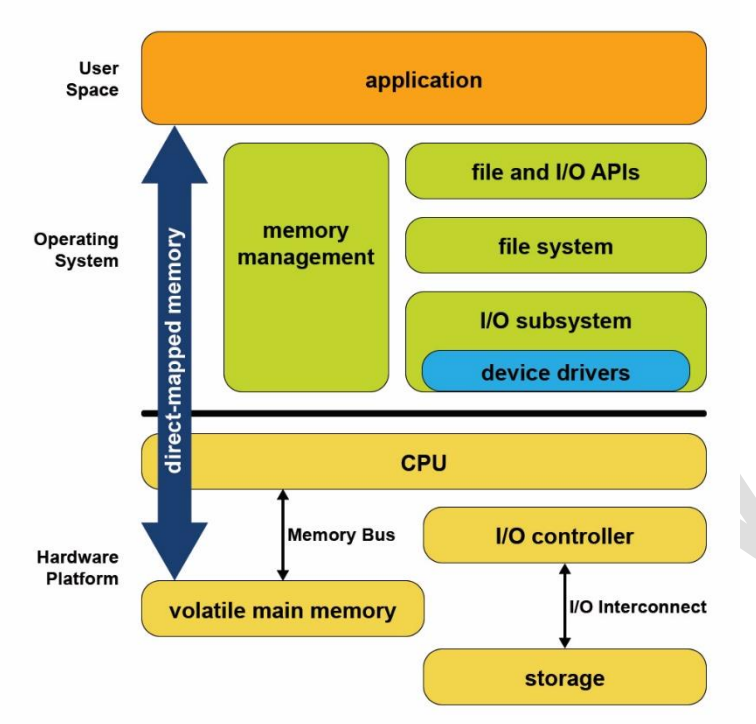

Figure 3-1: Storage and volatile memory in the operating system

The combination of direct application access to volatile memory combined with the operating system I/O access to storage devices supports the most common application programming model taught in introductory programming classes. In this model, developers allocate data structures and operate on them at byte granularity in memory. When the application wants to save data, it uses standard file API system calls to write the data to an open file. Within the operating system, the file system executes this write by performing one or more I/O operations to the storage device. Because these I/O operations are usually much slower than CPU speeds, it is typical for the operating system to suspend the application until the I/O completes.

Since persistent memory can be accessed directly by applications and can persist data in-place, it allows operating systems to support a direct programming model. Fortunately for application developers, while the first generation of persistent memory was under development, Microsoft Windows and Linux designers collaborated in the Storage and Networking Industry Association (SNIA) to define this new programming model. This is described in the SNIA NVM Programming Model Specification (https://www.snia.org/tech_activities/standards/curr_standards/npm).

Both Linux and Windows comply with this specification. As a result, the operating system support described in this chapter is common to at least the Windows and Linux operating systems. The programming model specification describes multiple ways that applications can use persistent memory along with the operating system extensions to support those programming options and are described in the remainder of this chapter.

Persistent Memory as Block Storage

The first operating system extension for persistent memory is the ability to detect the existence of persistent memory modules and load a device driver into the operating system's I/O subsystem as shown in Figure 3-2. This driver is called an NVDIMM driver and serves two important functions. First, it provides an interface for management and sysadmin utilities to configure and monitor the state of the persistent memory hardware and is similar to the functionality provided by storage device drivers.

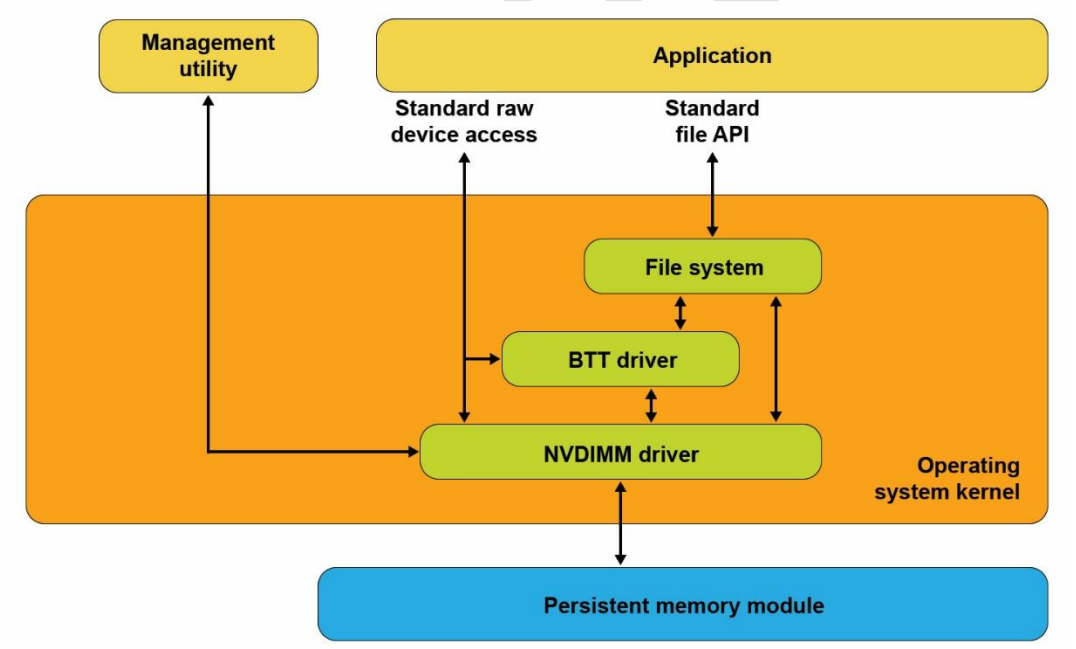

Figure 3-2 Persistent Memory as Block Storage

The NVDIMM driver presents persistent memory to applications and operating system modules as a fast block storage device. This means applications, file systems, volume managers and other storage middleware layers can use persistent memory the same way they use storage today, without modifications.

Figure 3-2 also shows a driver that can be optionally configured into the I/O subsystem called the Block Translation Table (BTT) driver. Storage devices such as HDDs and SSDs present a native block size with 512 and 4k bytes as two common native block sizes. Some storage devices, especially server-class storage, provide a guarantee that when a power failure or server failure occurs if a block write is in-flight, either all or none of the block will be written. The BTT driver is included to provide the same guarantee when using persistent memory as a block storage device. Most applications and files systems depend on this atomic write guarantee and should be configured to use the BTT driver although operating systems also provide the option to bypass the BTT driver for applications that implement their own protection against partial block updates.

Persistent Memory Aware File Systems

The next extension to the operating system is to make the file system aware of and optimized for persistent memory. File systems that have been extended for persistent memory include Linux ext4 and XFS, and Windows NTFS. As shown in Figure 3-3, these file systems can use the block driver in the I/O subsystem as described in the previous section or can bypass the I/O subsystem to directly use persistent memory as byte-addressable load/store memory as the fastest and shortest path to data stored in persistent memory. In addition to eliminating the I/O operation, this path enables small data writes to be executed faster than traditional block storage devices which require the file system to read the device's native block size, modify the block, then write the full block back to the device.

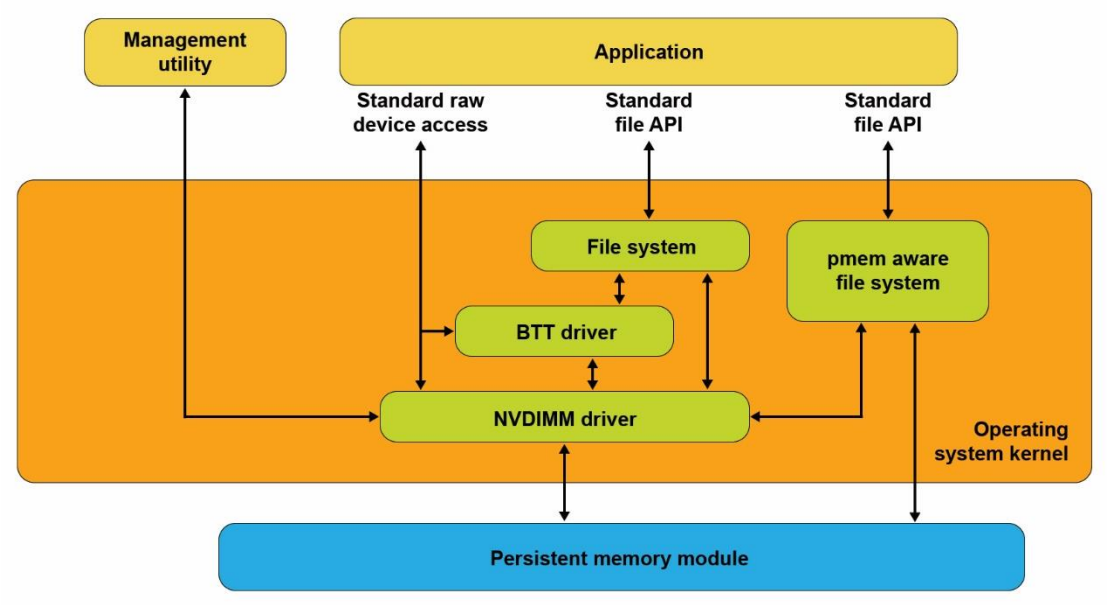

Figure 3-3 Persistent Memory Aware File System

These persistent memory aware file systems continue to present the familiar, standard file APIs to applications including the `open, close, read` and, `write` system calls. This allows applications to continue using this familiar file system APIs while benefiting from the higher performance of persistent memory.

Memory Mapped Files

Before describing the next operating system option for using persistent memory, we review memory mapped files in Linux and Windows. When memory mapping a file, the operating system allocates a portion of main memory, maps that memory into the application's virtual address space, and maintains a correlation between the portion of the file in memory, and the on-disk copy of the file. This allows an application to access and modify file data as byte-addressable in-memory data structures. This has the potential to improve performance and simplify application development, especially for applications that make frequent, small updates to file data.

Applications memory map a file by first opening the file, then passing the resulting file handle as a parameter to the `mmap()` system call in Linux or to `MapViewOfFile()` in Windows. Both return a pointer to the in-memory copy of a

portion of the file. Listing 3-1 shows an example of Linux C code that memory maps a file, write data into the file by accessing it like memory, and then uses the `msync` system call to perform the I/O operation to write the modified data to the file on the storage device. Listing 3-2 shows the equivalent operations on Windows. We will walk through and highlight the key steps in both of these code samples.

Listing 3-1 Memory mapped file on Linux Example

```
50 #include <err.h>
51 #include <fcntl.h>
52 #include <stdio.h>
53 #include <stdlib.h>
54 #include <string.h>
55 #include <sys/mman.h>
56 #include <sys/stat.h>
57 #include <sys/types.h>
58 #include <unistd.h>
59
60 int
61 main(int argc, char *argv[])
62 {
63    int fd;
64    struct stat stbuf;
65    char *pmaddr;
66
67    if (argc != 2) {
68         fprintf(stderr, "Usage: %s filename\n", argv[0]);
69         exit(1);
70    }
71
72    if ((fd = open(argv[1], O_RDWR)) < 0)
73         err(1, "open %s", argv[1]);
74
75    if (fstat(fd, &stbuf) < 0)
76         err(1, "stat %s", argv[1]);
77
78    /*
79     * Map the file into our address space for read & write.
80     * Use MAP_SHARED so stores are visible to other
programs.
81     */
```

```
82    if ((pmaddr = mmap(NULL, stbuf.st_size,
83                       PROT_READ|PROT_WRITE,
84                       MAP_SHARED, fd, 0)) == MAP_FAILED)
85          err(1, "mmap %s", argv[1]);
86
87    /* don't need the fd anymore, the mapping stays around */
88    close(fd);
89
90    /* store a string to the File */
91    strcpy(pmaddr, "This is new data written to the file");
92
93    /*
94     * Simplest way to flush is to call msync(). The length
95     * needs to be rounded up to a 4k page.
96     */
97    if (msync((void *)pmaddr, 4096, MS_SYNC) < 0)
98          err(1, "msync");
99
100   printf("Done.\n");
101   exit(0);
102 }
```

- Lines 67-73: We verify the caller passed a file name that can be opened. The open call will create the file if it does not already exist.
- Line 75: Here we retrieve the file statistics in order to use the length when we memory map the file.
- Line 82: Here we map the file into the application's address space to allow our program to access the contents as if in memory. In the second parameter, we pass the length of the file, requesting Linux to initialize memory with the full file. We also map the file with both READ and WRITE access and also as SHARED allowing other processes to map the same file.
- Line 88: We retire the file descriptor which is no longer needed once a file is mapped.
- Line 91: Here we write data into the file by accessing it like memory through the pointer returned by `mmap`.
- Line 97: Here we explicitly flush the newly-written string to the backing storage device.

Listing 3-2 shows an example of C code that memory maps a file, writes data into the file, then uses the FlushViewOfFile() and FlushFileBuffers() system calls to flush the modified data to the file on the storage device.

Listing 3-2 Memory mapped file on Windows Example

```
45  #include <fcntl.h>
46  #include <stdio.h>
47  #include <stdlib.h>
48  #include <string.h>
49  #include <sys/stat.h>
50  #include <sys/types.h>
51  #include <Windows.h>
52
53  int
54  main(int argc, char *argv[])
55  {
56          if (argc != 2) {
57                  fprintf(stderr, "Usage: %s filename\n",
argv[0]);
58                  exit(1);
59          }
60
61          /* Create the file or open if the file already
exists */
62          HANDLE fh = CreateFile(argv[1],
63                  GENERIC_READ|GENERIC_WRITE,
64                  0,
65                  NULL,
66                  OPEN_EXISTING,
67                  FILE_ATTRIBUTE_NORMAL,
68                  NULL);
69
70          if (fh == INVALID_HANDLE_VALUE) {
71                  fprintf(stderr, "CreateFile, gle: 0x%08x",
72                          GetLastError());
73                  exit(1);
74          }
75
76          /* Get the file length for use when memory mapping
later */
77          DWORD filelen = GetFileSize(fh, NULL);
```

```
78          if (filelen == 0) {
79                  fprintf(stderr, "GetFileSize, gle:
0x%08x",
80                          GetLastError());
81                  exit(1);
82          }
83
84          /* Create a file mapping object */
85          HANDLE fmh = CreateFileMapping(fh,
86                  NULL, /* security attributes */
87                  PAGE_READWRITE,
88                  0,
89                  0,
90                  NULL);
91
92          if (fmh == NULL) {
93                  fprintf(stderr, "CreateFileMapping, gle:
0x%08x",
94                          GetLastError());
95                  exit(1);
96          }
97
98          /* Map into our address space and get a pointer to
the beginning */
99          char *pmaddr = (char *)MapViewOfFileEx(fmh,
100                 FILE_MAP_ALL_ACCESS,
101                 0,
102                 0,
103                 filelen,
104                 NULL); /* hint address */
105
106          if (pmaddr == NULL) {
107                 fprintf(stderr, "MapViewOfFileEx, gle:
0x%08x",
108                         GetLastError());
109                 exit(1);
110          }
111
112          /* On windows must leave the file handle(s) open
while mmaped */
113
```

```
114          /* store a string to the beginning of the file
*/
115          strcpy(pmaddr, "This is new data written to the
file");
116
117          /* Flush this page with length rounded up to 4k
page size */
118          if (FlushViewOfFile(pmaddr, 4096) == FALSE) {
119                  fprintf(stderr, "FlushViewOfFile, gle:
0x%08x",
120                          GetLastError());
121                  exit(1);
122          }
123
124          /* Now flush the complete file to backing storage
*/
125          if (FlushFileBuffers(fh) == FALSE) {
126                  fprintf(stderr, "FlushFileBuffers, gle:
0x%08x",
127                          GetLastError());
128                  exit(1);
129          }
130
131          /* Explicitly unmap before closing the file */
132          if (UnmapViewOfFile(pmaddr) == FALSE) {
133                  fprintf(stderr, "UnmapViewOfFile, gle:
0x%08x",
134                          GetLastError());
135                  exit(1);
136          }
137
138          CloseHandle(fmh);
139          CloseHandle(fh);
140
141          printf("Done.\n");
142          exit(0);
143  }
```

- Lines 45-73: As in the previous example, we take the filename passed through argv and open the file.
- Line 77: We retrieve the file size to use later when memory mapping.

- Line 85: Here we take the first step to memory mapping a file by creating the file mapping. This step does not yet map the file into our application's memory space.
- Line 99: This step maps the file into our memory space.
- Line 115: As in the previous example, we write a string to the beginning of the file, accessing the file like memory.
- Line 118: We flush the modified memory page to the backing storage.
- Line 125: We flush the full file to backing storage, including any additional file metadata maintained by Windows.
- Line 132: We un-map the file and then close the file handles in lines 124 and 125.

Figure 3-4 shows what happens under the covers inside the operating system. When an application calls `mmap()` on Linux or `CreateFileMapping()` on Windows, the operating system allocates memory from its memory page cache, maps that memory into the application's address space, and creates the association with the file through a storage device driver.

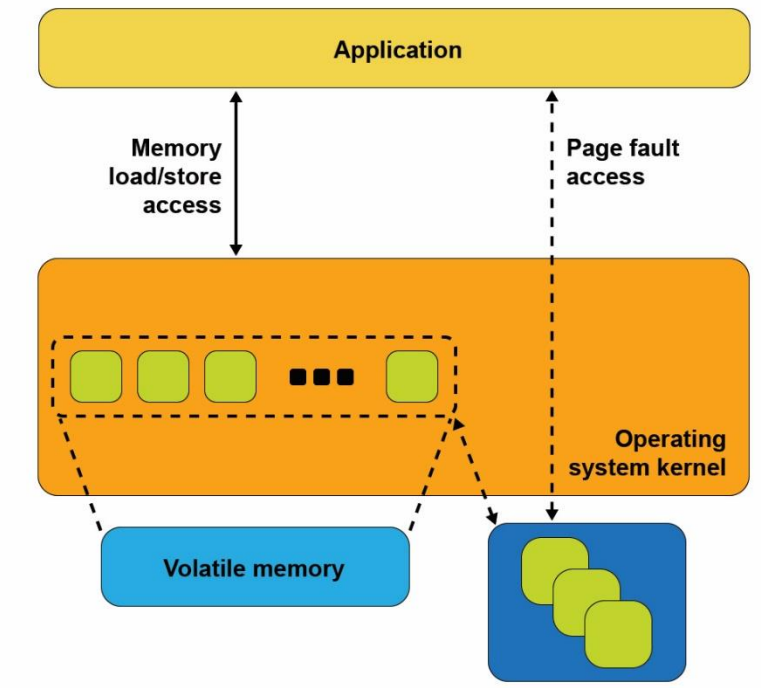

Figure 3-4 Memory mapped files with storage

As the application reads pages of the file in memory, if those pages are not present in memory, a page fault exception is raised to the operating system which will then read that page into main memory through storage I/O operations. The operating system also tracks writes to those memory pages and schedules asynchronous I/O operations to write the modifications back to the primary copy of the file on the storage device. Alternatively, if the application wants to ensure updates are written back to storage before continuing as we did in our code example, the `msync` system call on Linux or `FlushViewOfFile` on Windows executes the flush to disk. This may cause the operating system to suspend the program until the write finishes, similar to the file write operation described earlier.

This description of memory mapped files using storage highlights some of the disadvantages. Firstly, a portion of the limited kernel memory page cache in main memory is used to store a copy of the file. Secondly, for files that are larger than can fit in memory, the application may experience unpredictable and variable pauses as the operating system moves pages between memory and storage through I/O operations. Thirdly, updates to the in-memory copy are not persistent until written back to storage so can be lost in the event of a failure.

Persistent Memory Direct Access (DAX)

The persistent memory direct access feature in operating systems, referred to as DAX in Linux and Windows, uses the memory mapped file interfaces described in the previous section, but takes advantage of persistent memory's native ability to both store data and to be used as memory. Persistent memory can be natively mapped as application memory, eliminating the need for the operating system to cache files in volatile main memory.

To use DAX, the system administrator creates a filesystem on the persistent memory module and mounts that filesystem into the operating system's filesystem tree. For Linux users, persistent memory devices will appear as `/dev/pmem*` device special files. To show the persistent memory physical devices, system administrators can use the `ndctl` and `ipmctl` utilities shown in Listing 3-3 and 3-4.

Listing 3-3 Displaying persistent memory physical devices and regions on Linux

```
# ipmctl show -dimm

 DimmID | Capacity  | HealthState | ActionRequired | LockState | FWVersion
==============================================================================
 0x0001 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x0011 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x0021 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x0101 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x0111 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x0121 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x1001 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x1011 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x1021 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x1101 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x1111 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367
 0x1121 | 252.4 GiB | Healthy     | 0              | Disabled  | 01.02.00.5367

# ipmctl show -region

 SocketID | ISetID             | PersistentMemoryType | Capacity   | FreeCapacity | HealthState
=============================================================================================
 0x0000   | 0x2d3c7f48f4e22ccc | AppDirect            | 1512.0 GiB | 0.0 GiB      | Healthy
 0x0001   | 0xdd387f488ce42ccc | AppDirect            | 1512.0 GiB | 1512.0 GiB   | Healthy
```

Listing 3-4 Displaying persistent memory physical devices, regions, and namespaces on Linux

```
# ndctl list -DRN
{
  "dimms":[
    {
      "dev":"nmem1",
      "id":"8089-a2-1837-00000bb3",
      "handle":17,
      "phys_id":44,
      "security":"disabled"
    },
    {
      "dev":"nmem3",
      "id":"8089-a2-1837-00000b5e",
      "handle":257,
      "phys_id":54,
      "security":"disabled"
    },
    […snip…]
    {
      "dev":"nmem8",
      "id":"8089-a2-1837-00001114",
      "handle":4129,
```

```
      "phys_id":76,
      "security":"disabled"
    }
  ],
  "regions":[
    {
      "dev":"region1",
      "size":1623497637888,
      "available_size":1623497637888,
      "max_available_extent":1623497637888,
      "type":"pmem",
      "iset_id":-2506113243053544244,
      "mappings":[
        {
          "dimm":"nmem11",
          "offset":268435456,
          "length":270582939648,
          "position":5
        },
        {
          "dimm":"nmem10",
          "offset":268435456,
          "length":270582939648,
          "position":1
        },
        {
          "dimm":"nmem9",
          "offset":268435456,
          "length":270582939648,
          "position":3
        },
        {
          "dimm":"nmem8",
          "offset":268435456,
          "length":270582939648,
          "position":2
        },
        {
          "dimm":"nmem7",
          "offset":268435456,
          "length":270582939648,
          "position":4
        },
        {
          "dimm":"nmem6",
          "offset":268435456,
          "length":270582939648,
```

```
      "position":0
    }
  ],
  "persistence_domain":"memory_controller"
},
{
  "dev":"region0",
  "size":1623497637888,
  "available_size":0,
  "max_available_extent":0,
  "type":"pmem",
  "iset_id":3259620181632232652,
  "mappings":[
    {
      "dimm":"nmem5",
      "offset":268435456,
      "length":270582939648,
      "position":5
    },
    {
      "dimm":"nmem4",
      "offset":268435456,
      "length":270582939648,
      "position":1
    },
    {
      "dimm":"nmem3",
      "offset":268435456,
      "length":270582939648,
      "position":3
    },
    {
      "dimm":"nmem2",
      "offset":268435456,
      "length":270582939648,
      "position":2
    },
    {
      "dimm":"nmem1",
      "offset":268435456,
      "length":270582939648,
      "position":4
    },
    {
      "dimm":"nmem0",
      "offset":268435456,
      "length":270582939648,
```

```
          "position":0
        }
      ],
      "persistence_domain":"memory_controller",
      "namespaces":[
        {
          "dev":"namespace0.0",
          "mode":"fsdax",
          "map":"dev",
          "size":1598128390144,
          "uuid":"06b8536d-4713-487d-891d-795956d94cc9",
          "sector_size":512,
          "align":2097152,
          "blockdev":"pmem0"
        }
      ]
    }
  ]
}
```

When a filesystem is created and mounted using `/dev/pmem*` devices, they can be identified using the `df` command as shown in Listing 3-5.

Listing 3-5 Locating persistent memory on Linux

```
$ df -h /dev/pmem*
Filesystem      Size  Used Avail Use% Mounted on
/dev/pmem0      1.5T   77M  1.4T   1% /mnt/pmemfs0
/dev/pmem1      1.5T   77M  1.4T   1% /mnt/pmemfs1
```

Windows developers will use PowerShellCmdlets as shown in figure 3-6. In either case, assuming the administrator has granted you rights to create files, you can create one or more files in the persistent memory then memory map those files to your application using the same method as shown in code Listings 3-1 and 3-2.

Listing 3-6 Locating persistent memory on Windows

```
PS C:\Users\Administrator> Get-PmemDisk

Number Size   Health  Atomicity Removable Physical device IDs Unsafe shutdowns
------ ----   ------  --------- --------- ------------------- ----------------
2      249 GB Healthy None      True      {1}                 36
```

```
PS C:\Users\Administrator> Get-Disk 2 | Get-Partition

PartitionNumber  DriveLetter Offset   Size         Type
---------------  ----------- ------   ----         ----
1                            24576    15.98 MB     Reserved
2                D           16777216 248.98 GB    Basic
```

Managing persistent memory as files have several benefits. We can leverage the rich features of leading filesystems for organizing, managing, naming, and limiting access for users persistent memory files and directories. We can apply the familiar file system permissions, and access rights management for protecting data stored in persistent memory and for sharing persistent memory between multiple users. System administrators can use existing backup tools that rely on file system revision history tracking. Finally, application developers can build on existing memory mapping APIs as described above and applications that currently use memory mapped files, can use direct persistent memory without modifications.

Once a file backed by persistent memory is created and opened, an application still calls `mmap` or `MapViewOfFile` to get a pointer to the persistent media. The difference, as shown in Figure 3-5, is the persistent memory aware file system recognizes that the file is on persistent memory and programs the memory management unit (MMU) in the CPU to map the persistent memory directly into the application's address space. No copy in kernel memory is required, and no synchronizing to storage through I/O operations is required. The application can use the pointer returned by `mmap` or `MapViewOfFile` to operate on its data in-place directly in the persistent memory. Since no kernel I/O operations are required, and since the full file is mapped into the application's memory, it can manipulate large collections of data objects with higher and more consistent performance as compared to files on I/O-accessed storage.

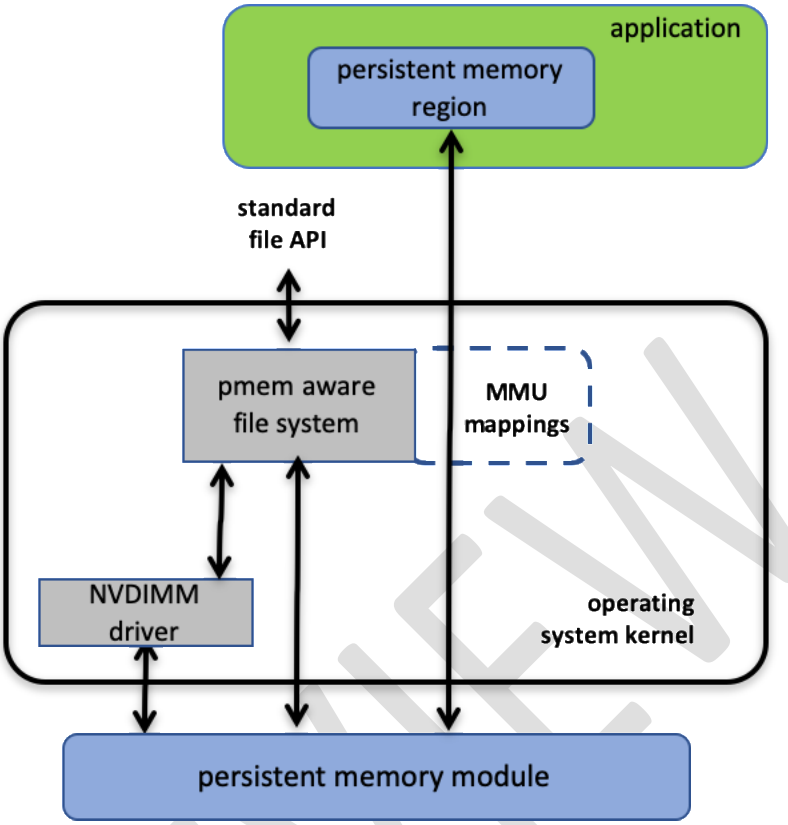

Figure 3-5 Direct Access (DAX) I/O and Standard File API I/O paths through the Kernel

Listing 3-4 shows a C source code example that uses DAX to write a string directly into persistent memory. In this example, we use one of the persistent memory API libraries included in Linux and Windows called `libpmem`. We describe these libraries in more detail in later chapters but will describe the use of two of the functions available in `libpmem` in the steps below. The APIs in `libpmem` are common across Linux and Windows and abstract the differences between underlying operating system APIs so this sample code is portable across both operating system platforms.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#ifndef _WIN32
```

```
39  #include <unistd.h>
40  #else
41  #include <io.h>
42  #endif
43  #include <string.h>
44  #include <libpmem.h>
45
46  /* using 4k of pmem for this example */
47  #define PMEM_LEN 4096
48
49  int
50  main(int argc, char *argv[])
51  {
52          char *pmemaddr;
53          size_t mapped_len;
54          int is_pmem;
55
56          if (argc != 2) {
57                  fprintf(stderr, "Usage: %s filename\n",
argv[0]);
58                  exit(1);
59          }
60
61          /* Create a pmem file and memory map it. */
62          if ((pmemaddr = pmem_map_file(argv[1], PMEM_LEN,
PMEM_FILE_CREATE,
63                                  0666, &mapped_len,
&is_pmem)) == NULL) {
64                  perror("pmem_map_file");
65                  exit(1);
66          }
67
68          /* Store a string to the persistent memory. */
69          char s[] = "This is new data written to the file";
70          strcpy(pmemaddr, s);
71
72          /* Flush our string to persistence. */
73          if (is_pmem)
74                  pmem_persist(pmemaddr, sizeof(s));
75          else
76                  pmem_msync(pmemaddr, sizeof(s));
77
```

```
78          /* Delete the mappings. */
79          pmem_unmap(pmemaddr, mapped_len);
80
81          printf("Done.\n");
82          exit(0);
83  }
```

Listing 3-4 DAX Programming Example

- Line 38: Here we handle differences between Linux and Windows include files.
- Line 44: We include the header file for the `libpmem` API used in this example.
- Line 49-59: We take the pathname argument like our previous examples.
- Line 62: The `pmem_map_file` function in libpmem handles opening the file and mapping it into our address space on both Windows and Linux. Since the file resides on persistent memory, the operating system programs the hardware memory management unit in the CPU to map the persistent memory region into our application's virtual address space. Pointer `pmemaddr` is set to the beginning of that region. The `pmem_map_file` function can also be used for memory-mapping disk-based files through kernel main memory as well as directly mapping persistent memory so `is_pmem` is set to `TRUE` if the file resides on persistent memory, and `FALSE` if mapped through main memory.
- Line 70: We write a string into persistent memory.
- Line 73: If the file resides on persistent memory, as in this example, the `pmem_persist` function uses the user-space machine instructions described in chapter 2 to ensure our string is flushed through CPU cache levels to the powerfail-safe domain and ultimately to persistent memory. If our file resided on disk-based storage, Linux `mmap` or Windows `FlushViewOfFile` would be used to flushed to storage. Note that we can pass small sizes here (the size of the string written is used in this example) instead of requiring flushes at page granularity when using `msync()` or `FlushViewOfFile()`.
- Line 79: Finally, we `unmap` the persistent memory region.

Summary

Figure 3-6 shows the complete view of the operating system support described in this chapter. As we discussed, an application can use persistent memory as a fast SSD, more directly through a persistent memory aware file system, or mapped directly into the application's memory space with the DAX option. DAX leverages operating system services for memory mapped files but takes advantage of the server hardware's ability to map persistent memory directly into the application's address space, avoiding the need to move data between main memory and storage. In the next few chapters, we will describe considerations for working with data directly in persistent memory and then discuss the APIs for simplifying development.

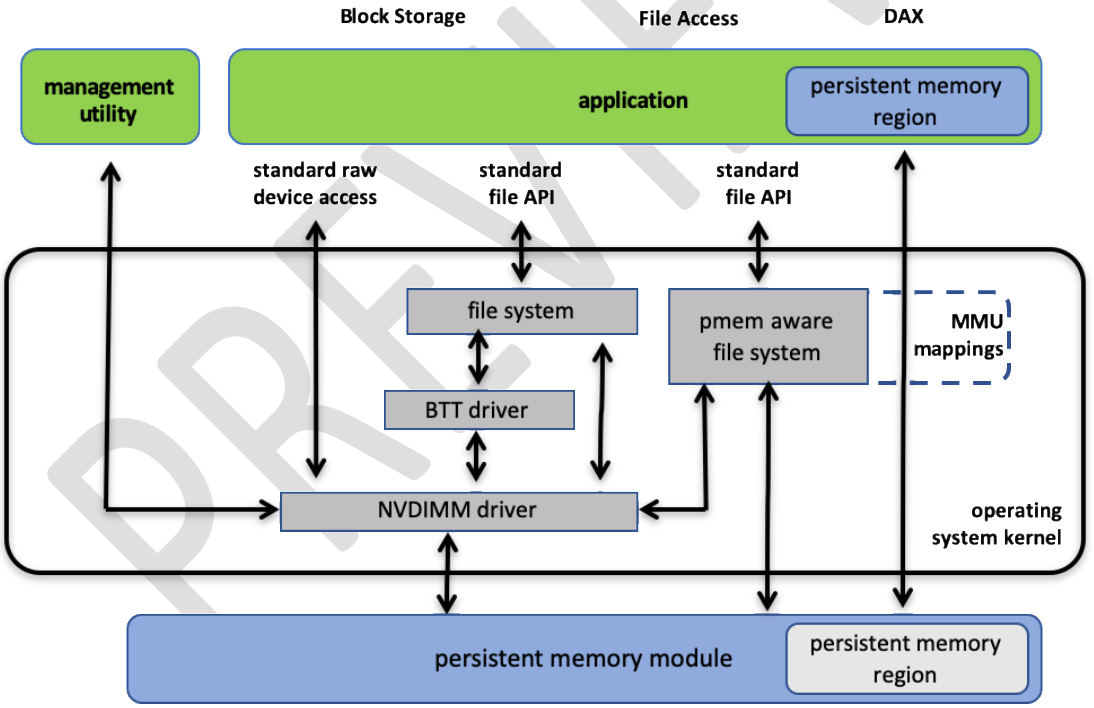

Figure 3-6 Persistent Memory Programming Options