

CHAPTER 3

Operating System Support for Persistent Memory

This chapter describes how operating systems manage persistent memory as a platform resource and describes the options they provide for applications to use persistent memory. We first compare 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 I/O controller. The operating system handles access to the storage through device driver modules loaded into the operating system's I/O subsystem.

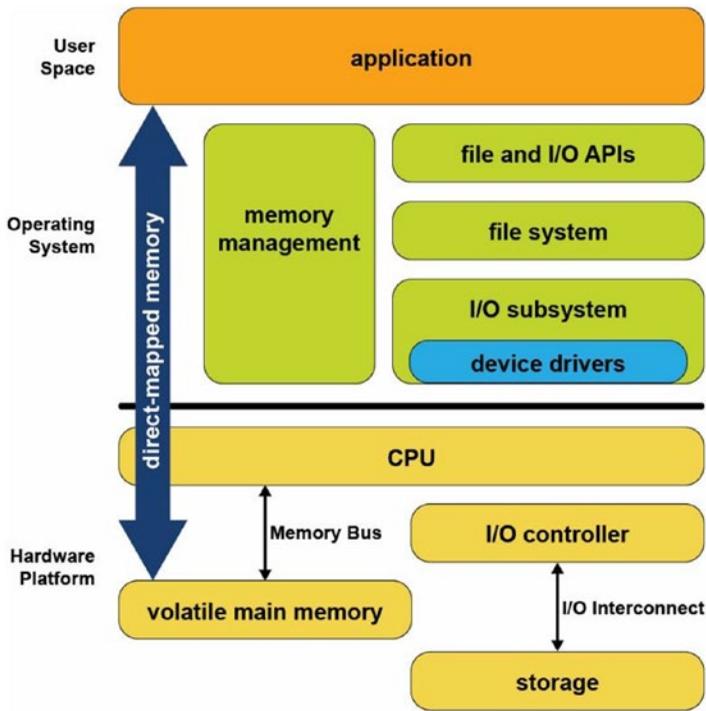


Figure 3-1. Storage and volatile memory in the operating system

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, the operating system typically suspends 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 new programming model that combines the performance of memory while persisting data like a non-volatile storage device. Fortunately for developers, while the first generation of persistent memory was under development, Microsoft Windows and Linux designers, architects and

developers collaborated in the Storage and Networking Industry Association (SNIA) to define a common programming model, so the methods for using persistent memory described in this chapter are available in both operating systems. More details can be found in the SNIA NVM programming model specification (https://www.snia.org/tech_activities/standards/curr_standards/npm).

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 NVDIMM driver serves two important functions. First, it provides an interface for management and system administrator utilities to configure and monitor the state of the persistent memory hardware. Second, it functions similarly to the storage device drivers.

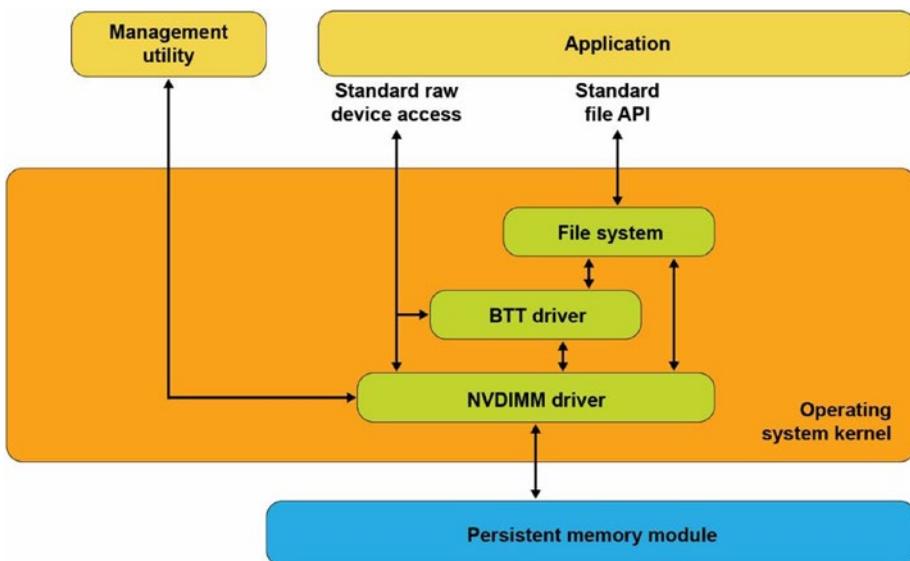


Figure 3-2. *Persistent memory as block storage*

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 the Block Translation Table (BTT) driver, which can be optionally configured into the I/O subsystem. Storage devices such as HDDs and SSDs present a native block size with 512k and 4k bytes as two common native block sizes. Some storage devices, especially NVM Express SSDs, provide a guarantee that when a power failure or server failure occurs while a block write is in-flight, either all or none of the block will be written. The BTT driver provides the same guarantee when using persistent memory as a block storage device. Most applications and file 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 be optimized for persistent memory. File systems that have been extended for persistent memory include Linux ext4 and XFS, and Microsoft Windows NTFS. As shown in Figure 3-3, these file systems can either use the block driver in the I/O subsystem (as described in the previous section) or 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 that require the file system to read the device's native block size, modify the block, and then write the full block back to the device.

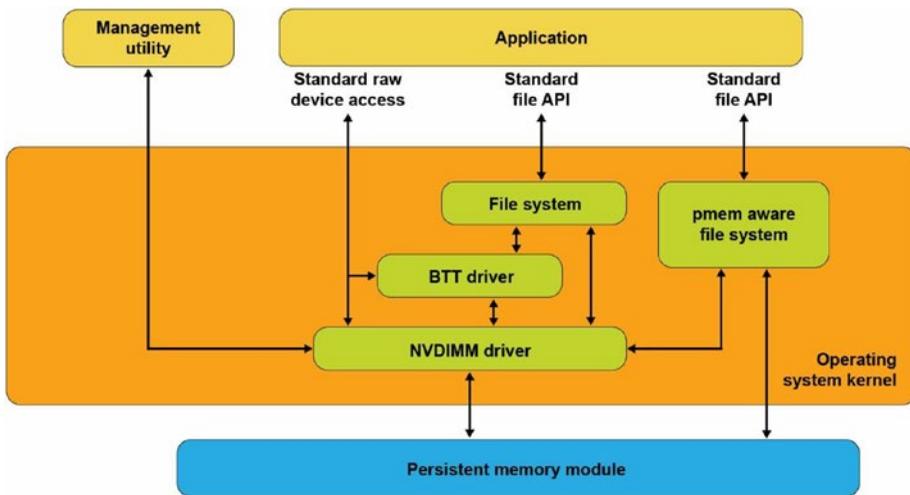


Figure 3-3. *Persistent memory-aware file system*

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 the familiar file APIs while benefiting from the higher performance of persistent memory.

Memory-Mapped Files

Before describing the next operating system option for using persistent memory, this section reviews memory-mapped files in Linux and Windows. When memory mapping a file, the operating system adds a range to the application's virtual address space which corresponds to a range of the file, paging file data into physical memory as required. 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, writes 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 walk through and highlight the key steps in both code samples.

Listing 3-1. `mmap_example.c` – 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",
69             argv[0]);
70         exit(1);
71     }
72
73     if ((fd = open(argv[1], O_RDWR)) < 0)
74         err(1, "open %s", argv[1]);
75
76     if (fstat(fd, &stbuf) < 0)
77         err(1, "stat %s", argv[1]);
78
79     /*

```

```

80     * Map the file into our address space for read
81     * & write. Use MAP_SHARED so stores are visible
82     * to other programs.
83     */
84     if ((pmaddr = mmap(NULL, stbuf.st_size,
85                       PROT_READ|PROT_WRITE,
86                       MAP_SHARED, fd, 0)) == MAP_FAILED)
87         err(1, "mmap %s", argv[1]);
88
89     /* Don't need the fd anymore because the mapping
90     * stays around */
91     close(fd);
92
93     /* store a string to the Persistent Memory */
94     strcpy(pmaddr, "This is new data written to the
95           file");
96
97     /*
98     * Simplest way to flush is to call msync().
99     * The length needs to be rounded up to a 4k page.
100    */
101    if (msync((void *)pmaddr, 4096, MS_SYNC) < 0)
102        err(1, "msync");
103
104    printf("Done.\n");
105    exit(0);
106 }

```

- Lines 67-74: 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 76: We retrieve the file statistics to use the length when we memory map the file.

- Line 84: 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 91: We retire the file descriptor which is no longer needed once a file is mapped.
- Line 94: We write data into the file by accessing it like memory through the pointer returned by `mmap`.
- Line 101: 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, and 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",
58             argv[0]);
59         exit(1);
60     }
61

```

```

62     /* Create the file or open if the file exists */
63     HANDLE fh = CreateFile(argv[1],
64         GENERIC_READ|GENERIC_WRITE,
65         0,
66         NULL,
67         OPEN_EXISTING,
68         FILE_ATTRIBUTE_NORMAL,
69         NULL);
70
71     if (fh == INVALID_HANDLE_VALUE) {
72         fprintf(stderr, "CreateFile, gle: 0x%08x",
73             GetLastError());
74         exit(1);
75     }
76
77     /*
78      * Get the file length for use when
79      * memory mapping later
80      */
81     DWORD filelen = GetFileSize(fh, NULL);
82     if (filelen == 0) {
83         fprintf(stderr, "GetFileSize, gle: 0x%08x",
84             GetLastError());
85         exit(1);
86     }
87
88     /* Create a file mapping object */
89     HANDLE fmh = CreateFileMapping(fh,
90         NULL, /* security attributes */
91         PAGE_READWRITE,
92         0,
93         0,
94         NULL);
95

```

```

96     if (fmh == NULL) {
97         fprintf(stderr, "CreateFileMapping,
98             gle: 0x%08x", GetLastError());
99         exit(1);
100    }
101
102    /*
103     * Map into our address space and get a pointer
104     * to the beginning
105     * */
106    char *pmaddr = (char *)MapViewOfFileEx(fmh,
107        FILE_MAP_ALL_ACCESS,
108        0,
109        0,
110        filelen,
111        NULL); /* hint address */
112
113    if (pmaddr == NULL) {
114        fprintf(stderr, "MapViewOfFileEx,
115            gle: 0x%08x", GetLastError());
116        exit(1);
117    }
118
119    /*
120     * On windows must leave the file handle(s)
121     * open while mmaped
122     * */
123
124    /* Store a string to the beginning of the file */
125    strcpy(pmaddr, "This is new data written to
126        the file");
127
128    /*
129     * Flush this page with length rounded up to 4K
130     * page size
131     * */

```

```

132     if (FlushViewOfFile(pmaddr, 4096) == FALSE) {
133         fprintf(stderr, "FlushViewOfFile,
134             gle: 0x%08x", GetLastError());
135         exit(1);
136     }
137
138     /* Flush the complete file to backing storage */
139     if (FlushFileBuffers(fh) == FALSE) {
140         fprintf(stderr, "FlushFileBuffers,
141             gle: 0x%08x", GetLastError());
142         exit(1);
143     }
144
145     /* Explicitly unmap before closing the file */
146     if (UnmapViewOfFile(pmaddr) == FALSE) {
147         fprintf(stderr, "UnmapViewOfFile,
148             gle: 0x%08x", GetLastError());
149         exit(1);
150     }
151
152     CloseHandle(fmh);
153     CloseHandle(fh);
154
155     printf("Done.\n");
156     exit(0);
157 }

```

- Lines 45-75: As in the previous Linux example, we take the file name passed through argv and open the file.
- Line 81: We retrieve the file size to use later when memory mapping.
- Line 89: 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 106: This step maps the file into our memory space.

- Line 125: As in the previous Linux example, we write a string to the beginning of the file, accessing the file like memory.
- Line 132: We flush the modified memory page to the backing storage.
- Line 139: We flush the full file to backing storage, including any additional file metadata maintained by Windows.
- Line 146-157: We unmap the file, close the file, then exit the program.

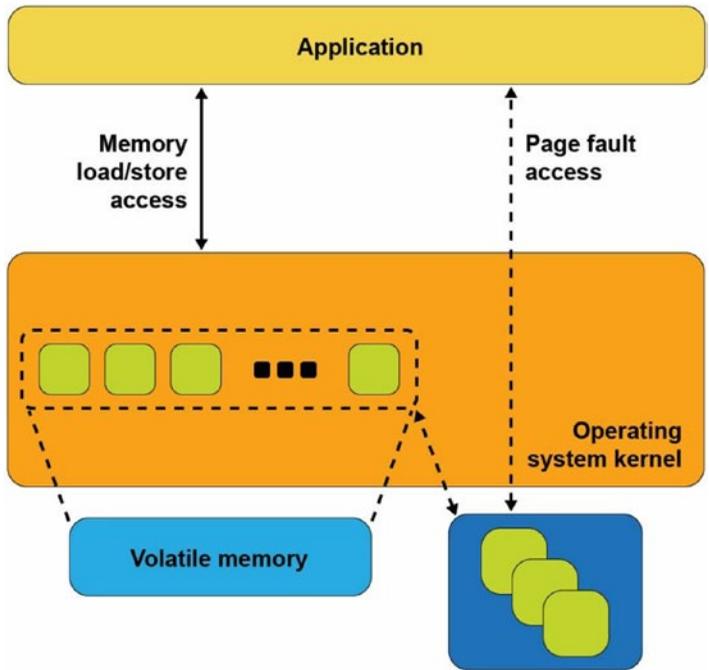


Figure 3-4. *Memory-mapped files with storage*

Figure 3-4 shows what happens 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.

As the application reads pages of the file in memory, and 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. First, a portion of the limited kernel memory page cache in main memory is used to store a copy of the file. Second, for files that cannot 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. Third, 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 file system on the persistent memory module and mounts that file system into the operating system's file system 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 Listings 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	FirmwareVersion
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 file system 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 Listing 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 and then memory map those files to your application using the same method 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 has several benefits:

- You can leverage the rich features of leading file systems for organizing, managing, naming, and limiting access for user's persistent memory files and directories.
- You 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.
- You can build on existing memory mapping APIs as described earlier and applications that currently use memory-mapped files and 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, shown in Figure 3-5, is that 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. Neither a copy in kernel memory nor 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 because 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.

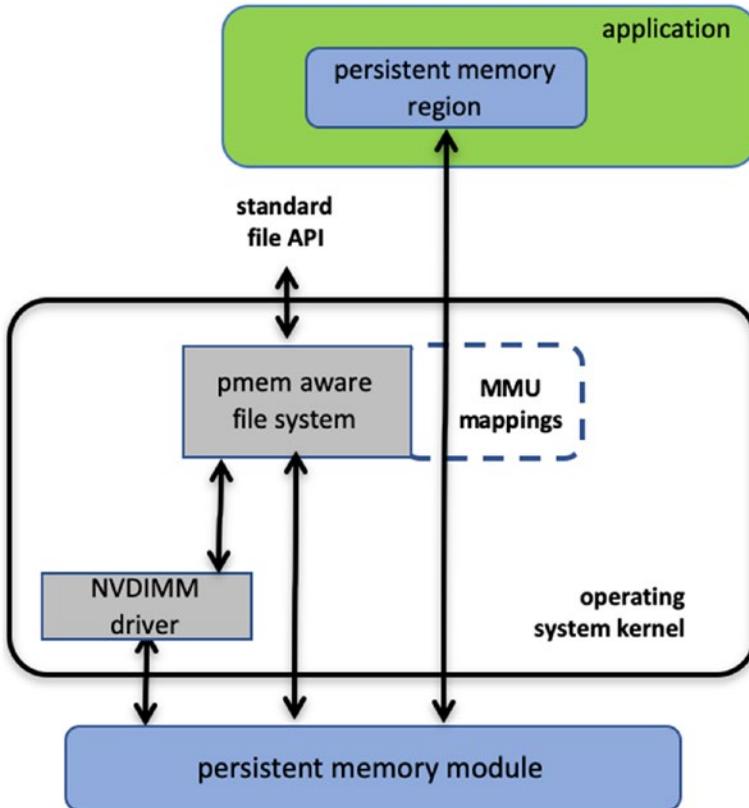


Figure 3-5. Direct access (DAX) I/O and standard file API I/O paths through the kernel

Listing 3-7 shows a C source code example that uses DAX to write a string directly into persistent memory. This example uses one of the persistent memory API libraries included in Linux and Windows called libpmem. Although we discuss these libraries in depth in later chapters, we describe the use of two of the functions available in libpmem in the following steps. 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.

Listing 3-7. DAX programming example

```

32 #include <sys/types.h>
33 #include <sys/stat.h>
34 #include <fcntl.h>
35 #include <stdio.h>
36 #include <errno.h>
37 #include <stdlib.h>
38 #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",
58             argv[0]);
59         exit(1);
60     }
61
62     /* Create a pmem file and memory map it. */
63     if ((pmemaddr = pmem_map_file(argv[1], PMEM_LEN,
64         PMEM_FILE_CREATE, 0666, &mapped_len,
65         &is_pmem)) == NULL) {

```

```

66     perror("pmem_map_file");
67     exit(1);
68 }
69
70 /* Store a string to the persistent memory. */
71 char s[] = "This is new data written to the file";
72 strcpy(pmемaddr, s);
73
74 /* Flush our string to persistence. */
75 if (is_pmem)
76     pmем_persist(pmемaddr, sizeof(s));
77 else
78     pmем_msync(pmемaddr, sizeof(s));
79
80 /* Delete the mappings. */
81 pmем_unmap(pmемaddr, mapped_len);
82
83 printf("Done.\n");
84 exit(0);
85 }

```

- Lines 38-42: We handle the differences between Linux and Windows for the include files.
- Line 44: We include the header file for the libpmem API used in this example.
- Lines 56-60: We take the pathname argument from the command line argument.
- Line 63-68: 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 MMU 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 72: We write a string into persistent memory.
- Lines 75-78: If the file resides on persistent memory, 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 power-fail safe domain and ultimately to persistent memory. If our file resided on disk-based storage, Linux `mmap` or Windows `FlushViewOfFile` would be used to flush 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 81: Finally, we unmap the persistent memory region.

Summary

Figure 3-6 shows the complete view of the operating system support that this chapter describes. 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. This avoids the need to move data between main memory and storage. The next few chapters describe considerations for working with data directly in persistent memory and then discuss the APIs for simplifying development.

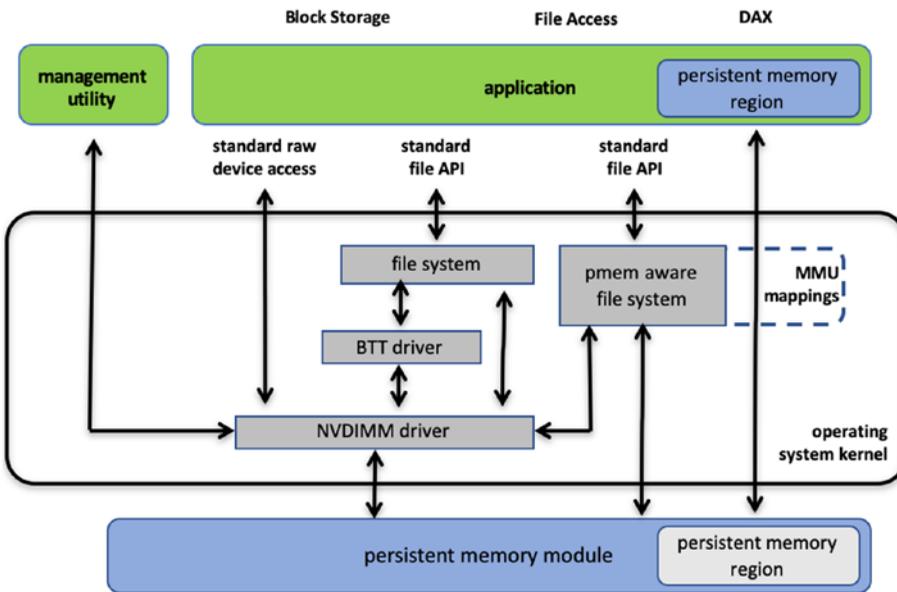


Figure 3-6. Persistent memory programming interfaces



Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.