CHAPTER 19

Advanced Topics

This chapter covers several topics that we briefly described earlier in the book but did not expand upon as it would have distracted from the focus points. The in-depth details on these topics are here for your reference.

Operating System Page Sizes

Operating systems manage memory in blocks known as pages. Although the default page size is determined by what the memory management unit (MMU) of the CPU supports, a page is commonly 4096 bytes (4KiB). The CPU MMU maintains a list of these pages referenced through a page table entry (PTE). The CPU needs to have Page Size Extension (PSE) capabilities, and nearly all current CPUs support this feature.

Imagine the following scenario when an application program requests 1 MiB of memory. To fulfill this request, an operating system must allocate 256 pages of 4KiB each. An overhead of approximately 1 KiB of memory is required for maintaining page directories and page tables.

When accessing this 1 MiB memory, each of the 256 page entries would be cached in the translation lookaside buffer (TLB). The TLB is a cache that maintains virtual address to physical address translations for faster lookup on subsequent memory requests. Populating the TLB with 256 entries for what could have been allocated in one single memory block is a disadvantage that can lead to a high TLB miss rate, resulting in lower performance. On systems with gigabytes or terabytes of memory, using 4KiB pages can result in millions of entries that can easily fill the TLB.

When the TLB fills up, a TLB entry will need to be freed to make way for the new entry. This can become a severe performance penalty if the TLB encounters a lot of thrashing generated by excessive eviction and loading of TLB entries. Using larger page sizes reduces TLB misses which results in better performance.

Pages are represented by PTEs; a Page Table Entry is part of a virtual memory hierarchy. This hierarchy gives virtual-to-physical address mappings at the page granularity. The x86_64 architectures use up to 64 bits to represent a single page, although in reality a page is represented by 48 bits. Even though bits 48 to 64 are discarded, they cannot be set to arbitrary values. Instead, all bits in this range have to be copies of bit 47 to keep addresses unique and allow future extensions like the 5-level page table. This is called sign-extension because it's very similar to the sign extension in two's complement. Figure 19-1 shows the sections represented by all 64 bits. Each section is used for a specific level in the virtual memory hierarchy shown in Figure 19-2.

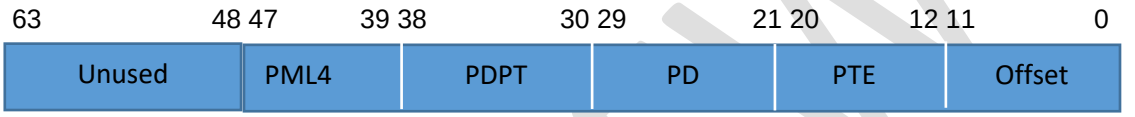

Figure 19-1. A 64-bit virtual address index layout.

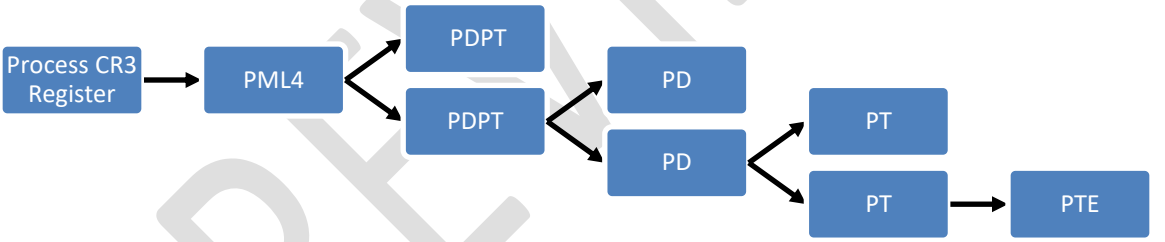

Figure 19-2. Data structures and pointers from CR3 register to a 4KiB page.

The physical address of the currently active level-4 page table, which is the root of the 4-level page table, is stored in the CR3 register. See this chain of pointers as we work through the layers:

Page Map Level → Page Directory Pointers → Page Directories → Page Tables → Page Table Entries

Each of the Page Map Level 4 (PML), Page Directory Pointer Table (PDPT), Page Directory (PD), and Page Table (PT) structures use a single 4KiB page. Each can store up to 512 8-byte entries.

To represent a 4KiB page requires one PML4, PDPD, PD, and PT entry. A 2MiB page requires one PML4, PDP, and PD entry. A 1GiB page requires only one PML4 and PDP entry.

To determine the ideal page size, you cannot look at the overhead of the paging structures alone. On Linux and Windows*, the virtual memory subsystem maintains separate data structures for each process. Although each data structure is just a few bytes per page, the accumulated memory footprint for thousands of processes on a system can grow quickly. For example, consider 1TiB shared memory database that has 100 processes—a small number when compared with most deployments. Table 19-1 calculates the kernel metadata overhead for a single process at 2,052MiB. For our database with 100 processes, this becomes 205,201MiB (~200GiB).

Table 19-1. Virtual memory metadata overhead per database process.

	# of 4KiB Pages	Size (MiB)
PML4	1	0.0039
PDPT	2	0.0078
PD	1024	4
PT	524288	2048
	Total	2052.0117
PTE	268435456	1048576
	Total	1050628.0117

Using Large and Huge Pages

A system can efficiently manage large amounts of memory in two ways:

1. Increase the number of page table entries in the CPU memory management unit.
2. Increase the page size.

Option 1 is expensive for the CPU manufacturer. A typical memory management in a processor usually supports hundreds or thousands of page table entries. Increasing the number of page table entries requires new CPU designs and manufacturing lines. This cost is usually passed on to the consumer. Additionally, hardware and memory management algorithms that work well with thousands of pages (megabytes of memory) may have difficulty performing well with millions (or even billions) of pages. This results in performance issues when an application needs to use more memory pages than the memory management unit supports; the system falls back to slower, software-based memory management, which causes the entire system to run more slowly.

Option 2 is implemented in most operating systems within the virtual memory subsystem. The PSE allows for page sizes of 2MiB (large page) and 1GiB (huge page) to co-exist alongside the default 4KiB pages. The page tables used by 2MB pages are suitable for managing multiple gigabytes of memory, and 1GB pages are best for scaling to terabytes of memory. Large-capacity persistent memory modules significantly increase a system's total amount of addressable memory into the multi-terabyte range. As a result, applications should be designed to utilize larger memory pages for optimal performance.

Applications such as databases are usually large- or huge-page aware. Their documentation commonly recommends that system and database administrators configure and use large pages for optimal performance when managing large amounts of memory. The disadvantage of using larger page sizes is potential internal fragmentation depending on how frequently pages are allocated and freed. Databases typically do not have a high page-churn rate that allows them to maintain predictable performance. The operating system virtual-memory subsystem handles multiple page sizes quite well unless the system is memory starved.

Most Linux and Windows releases support multiple page sizes. Look for "Large Page" or "Huge Page" support for your specific operating system distribution and version for instructions on how to enable and use them.

If our previous 1TiB database example used 2MiB pages instead of 4KiB, the overhead drops from ~200GiB to just ~0.5GiB. This is a significant memory saving and a huge reduction in the number of page table entries that the CPU MMU and kernel virtual memory needs to manage.

Transparent Huge Pages

Huge pages can be difficult to manage manually as the number of pages has to be known and configured before the application starts. Applications often require code changes to effectively use large or huge pages. Operating systems can assume some of this burden using transparent huge pages (THP). THP is an abstraction layer that automates most aspects of creating, managing, and using huge pages.

THP hides much of the complexity in using huge pages from system administrators and developers. Because the goal of THP is improving performance, its developers have tested and optimized it across a wide range of systems, configurations, applications, and workloads. This allows the default settings of THP to improve the performance of most system configurations. However, THP is not recommended for all workloads. Databases and Docker* are examples of where THP should be explicitly disabled.

Applications should be thoroughly tested with THP enabled and disabled to understand if this feature should be used. For more information about THP in Linux, read https://www.kernel.org/doc/Documentation/vm/transhuge.txt. Instructions for enabling and disabling THP are in the Linux kernel document and in your operating system user guide.

Non-Uniform Memory Access (NUMA)

NUMA is a computer memory design used in multiprocessing where the memory access time depends on the memory location relative to the processor. NUMA is used in a symmetric multiprocessing (SMP) system. An SMP system is a "tightly-coupled and share everything" system in which multiple processors working under a single operating system can access each other's memory over a common bus or "interconnect" path. With NUMA, a processor can access its own local memory faster than non-local memory (memory that is local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users.

CPU memory access is always fastest when the CPU can access its local memory. Typically, the CPU socket and the closest memory banks define a NUMA node. Whenever a CPU needs to access the memory of another NUMA node, it cannot

access it directly but is required to access it through the CPU owning the memory. Figure 19-3 shows a two-socket system with DRAM and persistent memory represented as “memory.”

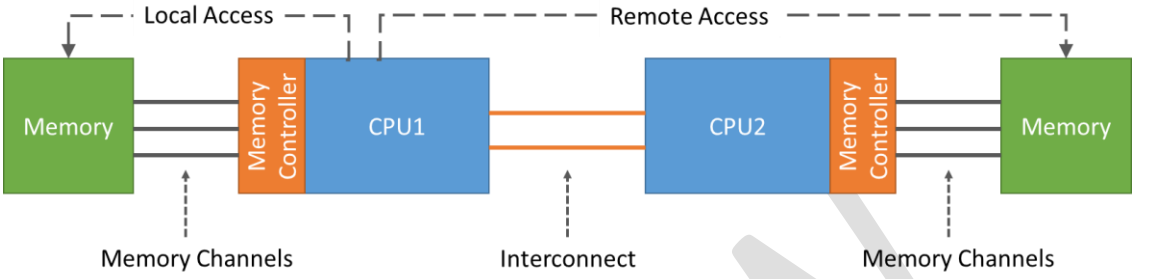

Figure 19-3. A two-socket CPU NUMA architecture showing local and remote memory access.

On a NUMA system, the greater the distance between a processor and a memory bank, the slower the processor's access to that memory bank. Performance-sensitive applications should therefore be configured so they allocate memory from the closest possible memory bank.

Performance-sensitive applications should also be configured to execute on a set number of cores, particularly in the case of multi-threaded applications. Because first-level caches are usually small, if multiple threads execute on one core, each thread will potentially evict cached data accessed by a previous thread. When the operating system attempts to multitask between these threads, and the threads continue to evict each other's cached data, a large percentage of their execution time is spent on cache line replacement. This issue is referred to as cache thrashing. We therefore recommend that you bind a multi-threaded application to a node rather than a single core, since this allows the threads to share cache lines on multiple levels (first-, second-, and last-level cache) and minimizes the need for cache fill operations. However, binding an application to a single core may be performant if all threads are accessing the same cached data. `numactl` allows you to bind an application to a particular core or NUMA node, and to allocate the memory associated with a core or set of cores to that application.

NUMACTL Linux Utility

On Linux we can use the `numactl` utility to display the NUMA hardware configuration and control which cores and threads application processes can run. The `libnuma`

library included in the `numactl` package offers a simple programming interface to the NUMA policy supported by the kernel. It is useful for more fine-grained tuning than the `numactl` utility. Further information is available in the `numa(7)` man page.

The `numactl --hardware` command displays an inventory of the available NUMA nodes within the system. The output shows only volatile memory, not persistent memory. We will show how to use the `ndctl` command to show NUMA locality of persistent memory in the next section. The number of NUMA nodes does not always equal the number of sockets. For example, an AMD Threadripper* 1950X has 1 socket and 2 NUMA nodes. The following output from `numactl` was collected from a two-socket Intel® Xeon® Platinum 8260L processor server with a total of 385GiB DDR4, 196GiB per socket.

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71
node 0 size: 192129 MB
node 0 free: 187094 MB
node 1 cpus: 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 72 73 74 75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95
node 1 size: 192013 MB
node 1 free: 191478 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

The `node distance` is a relative distance and not an actual time-based latency in nanoseconds or milliseconds.

`numactl` lets you bind an application to a particular core or NUMA node, and allocate the memory associated with a core or set of cores to that application. Some useful options provided by `numactl` are described in Table 19-2.

Table 19-2. numactl command options for binding processes to NUMA nodes or CPUs.

Option	Description
`--membind, -m`	Only allocate memory from specific NUMA nodes. The allocation will fail if there is not enough memory available on these nodes.

`--cpunodebind, -N`	Only execute the process on CPUs from the specified NUMA nodes.
`--physcpubind, -C`	Only execute process on the given CPUs.
`--localalloc, -l`	Always allocate on the current NUMA node.
`--preferred`	Preferably allocate memory on the specified NUMA node. If memory cannot be allocated, fall back to other nodes.

NDCTL Linux Utility

The `ndctl` utility is used to create persistent memory capacity for the operating system, called namespaces, as well as enumerating, enabling, and disabling the dimms, regions, and namespaces. Using the `-v` (verbose) option shows what NUMA node (`numa_node`) persistent memory DIMMS (-D), regions (`-R`), and namespaces (`-N`) belong to. Listing 19-1 shows the region and namespaces for a two socket system. We can correlate the `numa_node` with the corresponding NUMA node shown by the `numactl` command.

```
# ndctl list -Rv
{
  "regions":[
    {
      "dev":"region1",
      "size":1623497637888,
      "available_size":0,
      "max_available_extent":0,
      "type":"pmem",
      "numa_node":1,
      "iset_id":-2506113243053544244,
      "persistence_domain":"memory_controller",
      "namespaces":[
        {
          "dev":"namespace1.0",
          "mode":"fsdax",
          "map":"dev",
          "size":1598128390144,
          "uuid":"b3e203a0-2b3f-4e27-9837-a88803f71860",
          "raw_uuid":"bd8abb69-dd9b-44b7-959f-79e8cf964941",
```

```
            "sector_size":512,
            "align":2097152,
            "blockdev":"pmem1",
            "numa_node":1
          }
        ]
      },
      {
        "dev":"region0",
        "size":1623497637888,
        "available_size":0,
        "max_available_extent":0,
        "type":"pmem",
        "numa_node":0,
        "iset_id":3259620181632232652,
        "persistence_domain":"memory_controller",
        "namespaces":[
          {
            "dev":"namespace0.0",
            "mode":"fsdax",
            "map":"dev",
            "size":1598128390144,
            "uuid":"06b8536d-4713-487d-891d-795956d94cc9",
            "raw_uuid":"39f4abba-5ca7-445b-ad99-fd777f7923c1",
            "sector_size":512,
            "align":2097152,
            "blockdev":"pmem0",
            "numa_node":0
          }
        ]
      }
    ]
}
```

Listing 19-1. Region and namespaces for a two-socket system.

Intel® Memory Latency Checker Utility

To get absolute latency numbers between NUMA nodes on Intel® systems, you can use the Intel® Memory Latency Checker (Intel® MLC), available from https://software.intel.com/en-us/articles/intel-memory-latency-checker.

Intel MLC provides several modes specified through command line arguments:

- `--latency_matrix` prints a matrix of local and cross-socket memory latencies.
- `--bandwidth_matrix` prints a matrix of local and cross-socket memory bandwidths.
- `--peak_injection_bandwidth` prints peak memory bandwidths of the platform for various read-write ratios.
- `--idle_latency` prints the idle memory latency of the platform.
- `--loaded_latency` prints the loaded memory latency of the platform.
- `--c2c_latency` prints the cache to cache data-transfer latency of the platform.

Executing `mlc` or `mlc_avx512` with no arguments runs all the modes in sequence using the default parameters and values for each test and writes the results to the terminal. The following example shows running just the latency matrix on a two-socket Intel system.

```
# ./mlc_avx512 --latency_matrix -e -r
Intel(R) Memory Latency Checker - v3.6
Command line parameters: --latency_matrix -e -r

Using buffer size of 2000.000MiB
Measuring idle latencies (in ns)...
                Numa node
Numa node            0       1
       0          84.2   141.4
       1         141.5    82.4
```

- `--latency_matrix` prints a matrix of local and cross-socket memory latencies.
- `-e` means that the hardware prefetcher states do not get modified.
- `-r` is random access reads for latency thread.

MLC can be used to test persistent memory latency and bandwidth in either DAX or FSDAX modes. Commonly used arguments include:

- `-L` requests that large pages (2MB) be used (assuming they have been enabled).
- `-h` requests huge pages (1GB) for DAX file mapping.
- `-J` specifies a directory in which files for `mmap` will be created (by default no files are created). This option is mutually exclusive with `-j`.
- `-P` CLFLUSH is used to evict stores to persistent memory.

Examples:

Sequential read latency:

```
# mlc_avx512 --idle_latency -J/mnt/pmemfs
```

Random read latency:

```
# mlc_avx512 --idle_latency -l256 -J/mnt/pmemfs
```

NUMASTAT Utility

The `numastat` utility on Linux shows per NUMA node memory statistics for processors and the operating system. With no command options or arguments, it displays NUMA hit and miss system statistics from the kernel memory allocator. The default `numastat` statistics shows per-node numbers, in units of pages of memory, for example:

```
$ sudo numastat
                           node0           node1
numa_hit                8718076         7881244
numa_miss                     0               0
numa_foreign                  0               0
interleave_hit            40135           40160
local_node              8642532         2806430
other_node                75544         5074814
```

- `numa_hit` is memory successfully allocated on this node as intended.
- `numa_miss` is memory allocated on this node despite the process preferring some different node. Each `numa_miss` has a `numa_foreign` on another node.
- `numa_foreign` is memory intended for this node but is actually allocated on a different node. Each `numa_foreign` has a `numa_miss` on another node.
- `interleave_hit` is interleaved memory successfully allocated on this node as intended.
- `local_node` is memory allocated on this node while a process was running on it.
- `other_node` is memory allocated on this node while a process was running on another node.

Intel® VTune™ Amplifier - Platform Profiler

On Intel systems, you can use the Intel® VTune™ Amplifier - Platform Profiler (discussed in Chapter 15) to show CPU and memory statistics, including hit and miss rates of CPU caches and data accesses to DDR and persistent memory. It can also depict the system's configuration to show what memory devices are physically located on which CPU.

IPMCTL Utility

Persistent memory vendor- and server-specific utilities can also be used to show DDR and persistent memory device topology to help identify what devices are associated with which CPU sockets. For example, the `ipmctl show –topology` command displays the DDR and persistent memory (non-volatile) devices with their physical memory slot location (see Figure 19-4), if that data is available.

```
$ sudo  ipmctl show -topology

 DimmID | MemoryType                  | Capacity  | PhysicalID| DeviceLocat
==============================================================================
```

```
0x0001 | Logical Non-Volatile Device | 252.4 GiB | 0x0028    | CPU1_DIMM_A2
0x0011 | Logical Non-Volatile Device | 252.4 GiB | 0x002c    | CPU1_DIMM_B2
0x0021 | Logical Non-Volatile Device | 252.4 GiB | 0x0030    | CPU1_DIMM_C2
0x0101 | Logical Non-Volatile Device | 252.4 GiB | 0x0036    | CPU1_DIMM_D2
0x0111 | Logical Non-Volatile Device | 252.4 GiB | 0x003a    | CPU1_DIMM_E2
0x0121 | Logical Non-Volatile Device | 252.4 GiB | 0x003e    | CPU1_DIMM_F2
0x1001 | Logical Non-Volatile Device | 252.4 GiB | 0x0044    | CPU2_DIMM_A2
0x1011 | Logical Non-Volatile Device | 252.4 GiB | 0x0048    | CPU2_DIMM_B2
0x1021 | Logical Non-Volatile Device | 252.4 GiB | 0x004c    | CPU2_DIMM_C2
0x1101 | Logical Non-Volatile Device | 252.4 GiB | 0x0052    | CPU2_DIMM_D2
0x1111 | Logical Non-Volatile Device | 252.4 GiB | 0x0056    | CPU2_DIMM_E2
0x1121 | Logical Non-Volatile Device | 252.4 GiB | 0x005a    | CPU2_DIMM_F2
N/A    | DDR4                        | 32.0 GiB  | 0x0026    | CPU1_DIMM_A1
N/A    | DDR4                        | 32.0 GiB  | 0x002a    | CPU1_DIMM_B1
N/A    | DDR4                        | 32.0 GiB  | 0x002e    | CPU1_DIMM_C1
N/A    | DDR4                        | 32.0 GiB  | 0x0034    | CPU1_DIMM_D1
N/A    | DDR4                        | 32.0 GiB  | 0x0038    | CPU1_DIMM_E1
N/A    | DDR4                        | 32.0 GiB  | 0x003c    | CPU1_DIMM_F1
N/A    | DDR4                        | 32.0 GiB  | 0x0042    | CPU2_DIMM_A1
N/A    | DDR4                        | 32.0 GiB  | 0x0046    | CPU2_DIMM_B1
N/A    | DDR4                        | 32.0 GiB  | 0x004a    | CPU2_DIMM_C1
N/A    | DDR4                        | 32.0 GiB  | 0x0050    | CPU2_DIMM_D1
N/A    | DDR4                        | 32.0 GiB  | 0x0054    | CPU2_DIMM_E1
N/A    | DDR4                        | 32.0 GiB  | 0x0058    | CPU2_DIMM_F1
```

Figure 19-4. Topology report from the ipmctl show –topology command.

BIOS Tuning Options

The BIOS contains many tuning options that change the behavior of CPU, memory, persistent memory, and NUMA. The location and name may vary between server platform types, server vendors, persistent memory vendors, or BIOS versions. However, most applicable tunable options can usually be found in the Advanced menu under Memory Configuration and Processor Configuration. Refer to your system BIOS user manual for descriptions of each available option. You may want to test several BIOS options with the application(s) to understand which options bring the most value.

Automatic NUMA Balancing

Physical limitations to hardware are encountered when many CPUs and a lot of memory are required. The important limitation is the limited communication bandwidth between the CPUs and the memory. The NUMA architecture modification addresses

this issue. An application generally performs best when the threads of its processes are accessing memory on the same NUMA node as the threads are scheduled. Automatic NUMA balancing moves tasks (which can be threads or processes) closer to the memory they are accessing. It also moves application data to memory closer to the tasks that reference it. The kernel does this automatically when automatic NUMA balancing is active. Most operating systems implement this feature. This section discusses the feature on Linux; refer to your Linux distribution documentation for specific options as they may vary.

Automatic NUMA balancing is enabled by default in most, if not all, Linux distributions and will automatically activate at boot time when the operating system detects it is running on hardware with NUMA properties. To determine if the feature is enabled, use:

```
$ sudo cat /proc/sys/kernel/numa_balancing
```

A value of 1 (true) indicates the feature is enabled, whereas a value of 0 (zero/false) means it is disabled.

Automatic NUMA balancing uses several algorithms and data structures, which are only active and allocated if automatic NUMA balancing is active on the system, using a few simple steps:

- A task scanner periodically scans the address space and marks the memory to force a page fault when the data is next accessed.
- The next access to the data will result in a NUMA Hinting Fault. Based on this fault, the data can be migrated to a memory node associated with the thread or process accessing the memory.
- To keep a thread or process, the CPU it is using and the memory it is accessing together, the scheduler groups tasks that share data.

Manual NUMA tuning of applications using `numactl` will override any system-wide automatic NUMA balancing settings. Automatic NUMA balancing simplifies tuning workloads for high performance on NUMA machines. Where possible, we recommend statically tuning the workload to partition it within each node. Certain latency sensitive applications, such as databases, usually work best with manual configuration. However, in most other use cases, automatic NUMA balancing should help performance.

Using Volume Managers with Persistent Memory

We can provision persistent memory as a block device on which a file system can be created. Applications can access persistent memory using standard file APIs or memory map a file from the file system and access the persistent memory directly through load/store operations. The accessibility options are described in Chapters 2 and 3.

The main advantages of volume managers are increased abstraction, flexibility, and control. Logical volumes can have meaningful names like "databases" or "web." Volumes can be resized dynamically as space requirements change and migrated between physical devices within the volume group on a running system.

On NUMA systems, there is a locality factor between the CPU and the DRR and persistent memory that is directly attached to it. Accessing memory on a different CPU across the interconnect incurs a small latency penalty. Latency sensitive applications, such as databases, understand this and coordinate their threads to run on the same socket as the memory they are accessing.

Compared with SSD or NVMe capacity, persistent memory is relatively small. If your application requires a single file system that consumes all persistent memory on the system rather than one file system per NUMA node, a software volume manager can be used to create concatenations or stripes (RAID0) using all the system's capacity. For example, if you have 1.5TiB of persistent memory per CPU socket on a two-socket system, you could build a concatenation or stripe (RAID0) to create a 3TiB file system. If local system redundancy is more important than large file systems, mirroring (RAID1) persistent memory across NUMA nodes is possible. In general, replicating the data across physical servers for redundancy is better. Chapter 18 discusses remote persistent memory in detail, including using remote direct memory access (RDMA) for data transfer and replication across systems.

There are too many volume manager products to provide step-by-step recipes for all of them within this book. On Linux, you can use Device-Mapper (dmsetup), Multiple Device Driver (`mdadm`), and Linux Volume Manager (LVM) to create volumes that use the capacity from multiple NUMA nodes. Because most modern Linux distributions default to using LVM for their boot disks, we assume that you have some experience using LVM. There is extensive information and tutorials within the Linux documentation and on the web.

Figure 19-4 shows two regions on which we can create either an `fsdax` or `sector` type namespace that creates the corresponding `/dev/pmem0` and `/dev/pmem1` devices. Using `/dev/pmem[01]`, we can create an LVM physical volume which we then combine to create a volume group. Within the volume group, we are free to create as many logical volumes of the requested size as needed. Each logical volume can support one or more file systems.

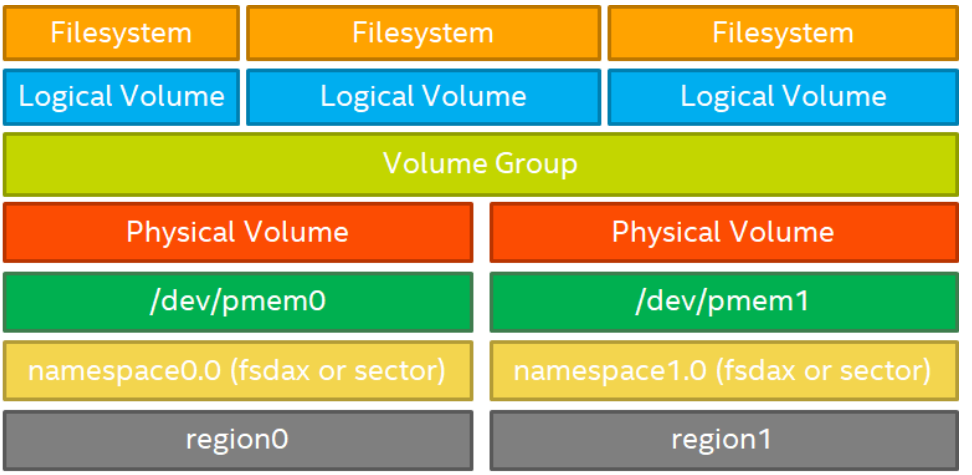

Figure 19-4. Linux Volume Manager architecture using persistent memory regions and namespaces.

We can also create a number of possible configurations if we were to create multiple namespaces per region, or partition the `/dev/pmem*` devices using `fdisk` or `parted`, for example. Doing this provides greater flexibility and isolation of the resulting logical volumes. However, if a physical NVDIMM fails, the impact is significantly greater since it would impact some or all of the file systems depending on the configuration.

Creating complex RAID volume groups may protect the data but at the cost of not efficiently using all the persistent memory capacity for data. Additionally, complex RAID volume groups do not support the DAX feature that some applications may require.

The mmap() MAP_SYNC Flag

Introduced in the Linux Kernel v4.15, the MAP_SYNC flag ensures that any needed file system metadata writes are completed before a process is allowed to modify directly mapped data. The MAP_SYNC was added to the `mmap()` system call to request the synchronous behavior, in particular, the guarantee provided by this flag is:

> *While a block is writeably mapped into page tables of this mapping, it is guaranteed to be visible in the file at that offset also after a crash.*

This means the file system will not silently relocate the block, and it will ensure that the file's metadata is in a consistent state so that the blocks in question will be present after a crash. This is done by ensuring that any needed metadata writes were done before the process is allowed to write pages affected by that metadata.

When a persistent memory region is mapped using MAP_SYNC, the memory-management code will check to see whether there are metadata writes pending for the affected file. However, it will not actually flush those writes out. Instead, the pages are mapped read-only with a special flag, forcing a page fault when the process first attempts to perform a write to one of those pages. The fault handler will then synchronously flush out any dirty metadata, set the page permissions to allow the write, and return. At that point, the process can write the page safely, since all the necessary metadata changes have already made it to persistent storage.

The result is a relatively simple mechanism that will perform far better than the currently available alternative of manually calling `fsync()` before each write to persistent memory. The additional IO from `fsync()` can potentially cause the process to block in what was supposed to be a simple memory write, introducing latency that may be unexpected and unwanted.

The `mmap(2)` man page in the Linux Programmer's manual describes the MAP_SYNC flag as follows:

> MAP_SYNC (since Linux 4.15)
> This flag is available only with the MAP_SHARED_VALIDATE mapping type; mappings of type MAP_SHARED will silently ignore this flag. This flag is supported only for files supporting DAX (direct mapping of persistent memory). For other files, creating a mapping with this flag results in an EOPNOTSUPP error.

Shared file mappings with this flag provide the guarantee that while some memory is writably mapped in the address space of the process, it will be visible in the same file at the same offset even after the system crashes or is rebooted. In conjunction with the use of appropriate CPU instructions, this provides users of such mappings with a more efficient way of making data modifications persistent.

Summary

In this chapter, we presented some of the more advanced topics for persistent memory including page size considerations on large memory systems, NUMA awareness and how it affects application performance, how to use volume managers to create DAX filesystems that span multiple NUMA nodes, and the `map_sync` flag for `mmap()`. Additional topics such as BIOS tuning were intentionally left out of this book as it is vendor and product specific. Performance and benchmarking of persistent memory products is left to external resources as there are too many tools – vdbench, sysbench, fio, etc – and too many options for each one, to cover in this book.