Concepts overview

The memory management in Linux is a complex system that evolved over the years and included more and more functionality to support a variety of systems from MMU-less microcontrollers to supercomputers. The memory management for systems without an MMU is called `nommu` and it definitely deserves a dedicated document, which hopefully will be eventually written. Yet, although some of the concepts are the same, here we assume that an MMU is available and a CPU can translate a virtual address to a physical address.

Virtual Memory Primer

The physical memory in a computer system is a limited resource and even for systems that support memory hotplug there is a hard limit on the amount of memory that can be installed. The physical memory is not necessarily contiguous; it might be accessible as a set of distinct address ranges. Besides, different CPU architectures, and even different implementations of the same architecture have different views of how these address ranges are defined.

All this makes dealing directly with physical memory quite complex and to avoid this complexity a concept of virtual memory was developed.

The virtual memory abstracts the details of physical memory from the application software, allows to keep only needed information in the physical memory (demand paging) and provides a mechanism for the protection and controlled sharing of data between processes.

With virtual memory, each and every memory access uses a virtual address. When the CPU decodes an instruction that reads (or writes) from (or to) the system memory, it translates the *virtual* address encoded in that instruction to a *physical* address that the memory controller can understand.

The physical system memory is divided into page frames, or pages. The size of each page is architecture specific. Some architectures allow selection of the page size from several supported values; this selection is performed at the kernel build time by setting an appropriate kernel configuration option.

Each physical memory page can be mapped as one or more virtual pages. These mappings are described by page tables that allow translation from a virtual address used by programs to the physical memory address. The page tables are organized hierarchically.

The tables at the lowest level of the hierarchy contain physical addresses of actual pages used by the software. The tables at higher levels contain physical addresses of the pages belonging to the lower levels. The pointer to the top level page table resides in a register. When the CPU performs the address translation, it uses this register to access the top level page table. The high bits of the virtual address are used to index an entry in the top level page table. That entry is then used to access the next level in the hierarchy with the next bits of the virtual address as the index to that level page table. The lowest bits in the virtual address define the offset inside the actual page.

Huge Pages

The address translation requires several memory accesses and memory accesses are slow relatively to CPU speed. To avoid spending precious processor cycles on the address translation, CPUs maintain a cache of such translations called Translation Lookaside Buffer (or TLB). Usually TLB is pretty scarce resource and applications with large memory working set will experience performance hit because of TLB misses.

Many modern CPU architectures allow mapping of the memory pages directly by the higher levels in the page table. For instance, on x86, it is possible to map 2M and even 1G pages using entries in the second and the third level page tables. In Linux such pages are called *huge*. Usage of huge pages significantly reduces pressure on TLB, improves TLB hit-rate and thus improves overall system performance.

There are two mechanisms in Linux that enable mapping of the physical memory with the huge pages. The first one is *HugeTLB filesystem*, or hugetlbfs. It is a pseudo filesystem that uses RAM as its backing store. For the files created in this filesystem the data resides in the memory and mapped using huge pages. The hugetlbfs is described at HugeTLB Pages.

Another, more recent, mechanism that enables use of the huge pages is called *Transparent HugePages*, or THP. Unlike the hugetlbfs that requires users and/or system administrators to configure what parts of the system memory should and can be mapped by the huge pages, THP manages such mappings transparently to the user and hence the name. See Transparent Hugepage Support for more details about THP.

Zones

Often hardware poses restrictions on how different physical memory ranges can be accessed. In some cases, devices cannot perform DMA to all the addressable memory. In other cases, the size of the physical memory exceeds the maximal addressable size of virtual memory and special actions are required to access portions of the memory. Linux groups memory pages into *zones* according to their possible usage. For example, ZONE_DMA will contain memory that can be used by devices for DMA, ZONE_HIGHMEM will contain memory that is not permanently mapped into kernel's address space and ZONE_NORMAL will contain normally addressed pages.

The actual layout of the memory zones is hardware dependent as not all architectures define all zones, and requirements for DMA are different for different platforms.

Nodes

Many multi-processor machines are NUMA - Non-Uniform Memory Access - systems. In such systems the memory is arranged into banks that have different access latency depending on the "distance" from the processor. Each bank is referred to as a *node* and for each node Linux constructs an independent memory management subsystem. A node has its own set of zones, lists of free and used pages and various statistics counters. You can find more details about NUMA in What is NUMA? and in NUMA Memory Policy.

Page cache

The physical memory is volatile and the common case for getting data into the memory is to read it from files. Whenever a file is read, the data is put into the *page cache* to avoid expensive disk access on the subsequent reads. Similarly, when one writes to a file, the data is placed in the page cache and eventually gets into the backing storage device. The written pages are marked as *dirty* and when Linux decides to reuse them for other purposes, it makes sure to synchronize the file contents on the device with the updated data.

Anonymous Memory

The *anonymous memory* or *anonymous mappings* represent memory that is not backed by a filesystem. Such mappings are implicitly created for program's stack and heap or by explicit calls to mmap(2) system call. Usually, the anonymous mappings only define virtual memory areas that the program is allowed to access. The read accesses will result in creation of a page table entry that references a special physical page filled with zeroes. When the program performs a write, a regular physical page will be allocated to hold the written data. The page will be marked dirty and if the kernel decides to repurpose it, the dirty page will be swapped out.

Reclaim

Throughout the system lifetime, a physical page can be used for storing different types of data. It can be kernel internal data structures, DMA'able buffers for device drivers use, data read from a filesystem, memory allocated by user space processes etc.

Depending on the page usage it is treated differently by the Linux memory management. The pages that can be freed at any time, either because they cache the data available elsewhere, for instance, on a hard disk, or because they can be swapped out, again, to the hard disk, are called *reclaimable*. The most notable categories of the reclaimable pages are page cache and anonymous memory.

In most cases, the pages holding internal kernel data and used as DMA buffers cannot be repurposed, and they remain pinned until freed by their user. Such pages are called *unreclaimable*. However, in certain circumstances, even pages occupied with kernel data structures can be reclaimed. For instance, in-memory caches of filesystem metadata can be re-read from the storage device and therefore it is possible to discard them from the main memory when system is under memory pressure.

The process of freeing the reclaimable physical memory pages and repurposing them is called (surprise!) *reclaim*. Linux can reclaim pages either asynchronously or synchronously, depending on the state of the system. When the system is not loaded, most of the memory is free and allocation requests will be satisfied immediately from the free pages supply. As the load increases, the amount of the free pages goes down and when it reaches a certain threshold (low watermark), an allocation request will awaken the `kswapd` daemon. It will asynchronously scan memory pages and either just free them if the data they contain is available elsewhere, or evict to the backing storage device (remember those dirty pages?). As memory usage increases even more and reaches another threshold - min watermark - an allocation will trigger *direct reclaim*. In this case allocation is stalled until enough memory pages are reclaimed to satisfy the request.

Compaction

As the system runs, tasks allocate and free the memory and it becomes fragmented. Although with virtual memory it is possible to present scattered physical pages as virtually contiguous range, sometimes it is necessary to allocate large physically contiguous memory areas. Such need may arise, for instance, when a device driver requires a large buffer for DMA, or when THP allocates a huge page. Memory *compaction* addresses the fragmentation issue. This mechanism moves occupied pages from the lower part of a memory zone to free pages in the upper part of the zone. When a compaction scan is finished free pages are grouped together at the beginning of the zone and allocations of large physically contiguous areas become possible.

Like reclaim, the compaction may happen asynchronously in the `kcompactd` daemon or synchronously as a result of a memory allocation request.

OOM killer

It is possible that on a loaded machine memory will be exhausted and the kernel will be unable to reclaim enough memory to continue to operate. In order to save the rest of the system, it invokes the *OOM killer*.

The *OOM killer* selects a task to sacrifice for the sake of the overall system health. The selected task is killed in a hope that after it exits enough memory will be freed to continue normal operation.