

1 Slide '*kernel-level-memory-management*'

1.1 Page Descriptor

In Linux, **state information of a page frame is kept in a page descriptor** of type `struct page` (or `struct mem_map_t`), and all page descriptors, which are 32 byte long, are stored in an array called `mem_map` (the space required by it is slightly less than 1% of the whole RAM). These data structures are defined into `include/linux/mm.h`.

The `virt_to_page(addr)` macro yields the address of the page descriptor associated with the linear address `addr`.

`struct page` has many fields but the most important are:

`atomic_t _count` It represent a usage reference counter for the page. If it is set to -1, the corresponding page frame is free and can be assigned to any process or to the kernel itself. If it is set to a value greater than or equal to 0, the page frame is assigned to one or more processes or is used to store some kernel data structures. The `page_count()` function returns the value of the `_count` field increased by one, that is, the number of users of the page. This field is managed via atomic updates, such as with `LOCK` directives.

`struct list_head lru` Contains pointers to the least recently used doubly linked list of pages.

`unsigned long flags` Array of flags used to describe the status of current page frame (but also encodes the zone number to which the page frame belongs). There are up to 32 flags and Linux kernel defines many macros to manipulate them. Some flags are:

`PG_locked` The page is locked; for instance, it is involved in a disk I/O operation.

`PG_dirty` The page has been modified.

`PG_reserved` The page frame is reserved for kernel code or is unusable.

1.2 Free list

Linux uses **free list** to manage memory allocation. **It operates by connecting unallocated regions of memory together in a linked list, using the first word of each unallocated region as a pointer to the next.**

Free lists make the allocation and deallocation operations very simple. **To free a region, one would just link it to the free list. To allocate a region, one would simply remove a single region from the end of the free list and use it.**

1.3 NUMA

Is extremely important to remember that Linux 2.6 supports the *Non-Uniform Memory Access* (NUMA) model, **in which the access times for different memory locations from a given CPU may vary** and, according to that architecture, physical memory is partitioned in several **nodes**. The time needed by a given CPU to access pages within a single node is the same. However, this time might not be the same for two different CPUs.

1.4 NUMA Node Descriptor

Be careful that Linux splits physical memory inside each node into several zones. We have 3 free lists of frames, depending on the frame positioning within available zones (defined in `include/linux/mmzone.h`) which are:

`ZONE_DMA` Contains page frames of memory below 16 MB, that is page frames that can be used by old ISA-based devices (*Direct Memory Access* (DMA) processors).

`ZONE_NORNMAL` Contains page frames of memory at and above 16 MB and below 896 MB (direct mapped by the kernel).

`ZONE_HIGHMEM` Contains page frames of memory at and above 896 MB (only page cache and user).

To represent a NUMA node, Linux uses a descriptor of type `struct pg_data_t`. All node descriptors are stored in a singly linked list, whose first element is pointed to by the `pgdat_list` variable. Be careful to the fact that this data structure is used by Linux kernel even if the architecture is based on *Uniform Memory Access* (UMA): in fact Linux makes use of a single node that includes all system physical memory. Thus, the `pgdat_list` variable points to a list consisting of a single element (node 0) stored in the `contig_page_data` variable.

Remember that free lists information is kept within the `struct pg_data_t` data structure. In fact the most important fields of `struct pg_data_t` are:

`struct page *node_mem_map` Array of page descriptors of the node

`struct zone [] node_zones` Array of zone descriptors of the node

1.5 Zone Descriptor

Obliviously each memory zone has its own descriptor of type `struct zone` and many fields of this data structure are used for page frame reclaiming. However, most important fields are:

`struct page * zone_mem_map` Pointer to first page descriptor of the zone.

`spinlock_t lock` Spin lock protecting the descriptor.

`struct free_area [] free_area` Identifies the blocks of free page frames in the zone

In summary, Linux has links to the memory node and to the zone inside the node that includes the corresponding page frame of type `struct page`.

1.6 Buddy allocator

The technique adopted by Linux to solve the external fragmentation problem is based on the well-known **buddy system** algorithm. All free page frames are grouped into 11 lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 contiguous page frames, respectively. The largest request of 1024 page frames corresponds to a chunk of 4 MB of contiguous RAM. We use the term **order to indicate the logarithmic size of a block**.

Assume there is a request for a group of 256 contiguous page frames (i.e., one megabyte). The algorithm checks first to see whether a free block in the 256-page-frame list exists. If there is no such block, the algorithm looks for the next larger block — a free block in the 512-page-frame list. If such a block exists, the kernel allocates 256 of the 512 page frames to satisfy the request and inserts the remaining 256 page frames into the list of free 256-page-frame blocks. If there is no free 512-page block, the kernel then looks for the next larger block (i.e., a free 1024-page-frame block). If such a block exists, it allocates 256 of the 1024 page frames to satisfy the request, inserts the first 512 of the remaining 768 page frames into the list of free 512-page-frame blocks, and inserts the last 256 page frames into the list of free 256-page-frame blocks. If the list of 1024-page-frame blocks is empty, the algorithm gives up and signals an error condition.

Linux 2.6 uses a different buddy system for each zone. Thus, in the x86 architecture, there are **3 buddy systems**: the first handles the page frames suitable for ISA DMA, the second handles the "normal" page frames, and the third handles the high memory page frames. Each buddy system relies on the following main data structures:

- **The `mem_map` array where all page descriptors are stored.** Actually, each zone is concerned with a subset of the `mem_map` elements. The first element in the subset and its number of elements are specified, respectively, by the `zone_mem_map` and `size` fields of the zone descriptor.
- The array consisting of eleven elements of type `struct free_area`, one element for each group size. As we said the array is stored in the `free_area` field of the zone descriptor.

Let us consider the k^{th} element of the `struct free_area` array in the zone descriptor, which identifies all the free blocks of size 2^k . In this data structure there is a pointer of type `struct list_head` which is the head of a doubly linked circular list that collects the page descriptors associated with the free blocks of 2^k pages. Besides the head of the list, the k^{th} element of the `struct free_area` array includes also the field `nr_free`, which specifies the number of free blocks of size 2^k pages, and a pointer to a bitmap that keeps fragmentation information.

Recall that spin locks are used to manage `mem_map` AND `struct free_area` array.

To achieve better performance a little number of page frames are kept in cache to quickly satisfy the allocation requests for single page frames.

1.7 API

Page frames can be requested by using some different functions and macros (APIs) (they return NULL in case of failure, a linear address of the first allocated page in case of success) which prototype are stored into `#include <linux/malloc.h>`. The most important are:

`get_zeroed_page(gfp_mask)` Function used to obtain a page frame filled with zeros.

`__get_free_page(gfp_mask)` Macro used to get a single page frame.

`__get_free_pages(gfp_mask, order)` Macro used to request 2^{order} contiguous page frames returning the linear address of the first allocated page.

`free_page(addr)` This macro releases the page frame having the linear address `addr`.

The parameter `gfp_mask` is a group of flags that specify **how to look for free page frames** and they are extremely important when we require page frame allocation in different contexts including:

Interrupt context allocation is requested by an **interrupt handler** which uses above function with `GFP_ATOMIC` flag (equivalent to `__GFP_HIGH`) **which means that the kernel is allowed to access the pool of reserved page frames: therefore the call cannot lead to sleep (that is no wait)** An atomic request never blocks: if there are not enough free pages the allocation simply fails.

Process context allocation is caused by a system call using `GFP_KERNEL` or `GFP_USER` (both equivalent to `__GFP_WAIT | __GFP_IO | __GFP_FS`) according to which kernel is allowed to block the current process waiting for free page frames (`__GFP_WAIT`) and to perform I/O transfers on low memory pages in order to free page frames (`__GFP_IO`): therefore the call can lead to sleep.

2 Part 1

Since a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter is overcoming these limitations. A dependence can be overcome in two different ways: maintaining the dependence but avoiding a hazard, and eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

Top and Bottom Halves One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind. Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt—the one you register with `request_irq`. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time. The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this operation is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

ksoftirqd is a per-cpu kernel thread which runs in background (as a daemon): is triggered to handle the software interrupts in process context.

`NR_CPUS` structures (the default value for this macro is 32; it denotes the maximum number of CPUs in the system)