1 Slide *'hardware-insights'*

1.1 OOO Pipeline execution and imprecise exceptions

When processor executes instructions in an order governed by the availability of input data and execution units, rather than by their original order in a program, we are adopting an **out-of-order execution paradigm**; **in other words different instructions can surpass each other depending on data or micro-controller availability.** We distinguish two events when we use this kind of paradigm: the **emission**, that is the action of injecting instructions into the pipeline; the **retire**, that is the action of committing instructions and making their side effects visible in terms of ISA exposed architectural resources

Is important to recall that out-of-order completion must preserve exception behaviour in the sense that exactly those exceptions that would arise if the program were executed in strict program order actually do arise. However, when we use OOO execution paradigm a processor may generate the so called **imprecise exceptions**. **An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order**. In other words imprecise exceptions can occur because when:

- The pipeline may have already completed instructions that are later in program order than the instruction causing the exception.
- The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception.

Recall that any instruction may change the micro-architectural state, although finally not committing its actions onto ISA exposed resources. Since the the pipeline may have not yet completed the execution of instructions preceding the offending one, hardware status can been already changed an this fact can be exploited by several attacks (like **Meltdown**).

1.2 Tomasulo algorithm

Tomasulo's algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution and enables more efficient use of multiple execution units. Suppose two operation A and B such that A precedes B in program order, that algorithm permit to resolve three hazard:

RAW (Read After Write) B reads a datum before A writes it.

WAW (Write After Write) B writes a datum before A writes the same datum.

WAR (Write After Read) B writes a datum before A reads the same datum.

RAW hazards are avoided by executing an instruction only when its operands are available, while WAR and WAW hazards are eliminated by *register renaming*

1.3 UMA

When we have a **single main memory** that has a *symmetric relationship to all processors and a uniform access time from any processor*, these multiprocessors are most often called *symmetric shared-memory multiprocessors* (**SMPs**), and this style of architecture is sometimes called *uniform memory access* (**UMA**): in fact, **all processors have a uniform latency from memory**. **The term shared memory refers to the fact that the address space is shared; that is, the same physical address on two processors refers to the same location in memory.** In this architecture all CPUs can have one or more level of cache. However this architecture is obliviously **not** scalable when the number of CPUs grows.

1.4 NUMA

When we have a distributed memory, a multiprocessor architecture is usually called *distributed shared-memory* (**DSM**). When we use this kind of system, we have two benefits:

- A cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node.
- Reduces the latency for accesses to the local memory by a CPU.

The key disadvantages for that architecture is that communicating data between processors becomes more complex, **and that it requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories.**

The DSM multiprocessors are also called **NUMAs** (*non-uniform memory access*), **since the access time depends on the location of a data word in memory.** In fact, when a CPU wants to access to an item stored into his node, performing a *local access* involving inner private/shared caches and controllers, access latency is very low. However when a CPU wants to access to an item stored on another node, performing a *remote accesses* involving remote controllers and caches, latency can be very high respect to previous case.

1.5 The problem of cache coherence

Unfortunately, **the view of memory held by different processors is through their individual caches**, which, without any additional precautions, could end up seeing different values of a same shared data (**cache coherence** problem).

By definition, **coherence defines what values can be returned by a read** (a **cache coherence protocols** defines how to maintain coherence) while **consistency determines when a written value will be returned by a read** (a **memory consistency protocol** defines when written value must be seen by a reader).

A memory system is coherent if:

1. A read from location X, previously written by a processor, returns the last written value if no other processor carried out writes on X in the meanwhile. **This property preserve program order that is the causal consistency along program order.**

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses. **This property assure that a processor couldn't continuously read an old data value (Avoidance of staleness).**

3. **Writes to the same location are serialized**; that is, two writes to the same location by any two processors are seen in the same order by all processors.

The choice and the design of a coherence protocol depends on many factors including: overhead, latency, cache policies, interconnection topology and so on. **However the Key to implementing a cache coherence protocol is tracking the state of any copy of a data block**. There are two classes of protocol which define when update aforementioned copies:

Update protocol When we use this type of protocol, also called *write update* or *write broadcast*, when a core writes to a block, it updates all other copies (**it consumes considerably more bandwidth**).

Invalidate protocol When we use this type of protocol, a processor has **exclusive access** to a data item before it writes that item; moreover that CPU invalidates other copies on a write that is no other readable or writeable copies of an item exist when the write occurs. **It is the most common protocol, but suffer of some latency.**

1.6 Snooping protocol

The key to implementing an invalidate protocol is the use of the bus, or another broadcast medium, called *network* to perform invalidates and to issue "transactions" on the state of cache blocks.

To perform any operation, the processor simply **acquires** bus access and broadcasts the address to be invalidated on the bus. All processors continuously **snoop** on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated. **A state transition cannot occur unless the broadcast medium is acquired by the source controller and are carried out atomically with a distribute fashions thanks to *serialization* over the broadcast medium**.

When we perform a read, we also need to locate a data item when a cache miss occurs. In a **write-through cache**, it is easy to find the recent value of a data item, *since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched* (using write through simplifies the implementation of cache coherence). For a **write-back cache**, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a cache rather than in memory (the CPU must get data from another cache)

2 Slide *'kernel-programming-basics'*

2.1 Segmentation

Intel microprocessors perform address translation in **three** different ways:

Real Mode This mode exists mostly to **maintain processor compatibility with older models and to allow the operating system to bootstrap**. In this modality, a logical address is composed of a `seg` segment (hold by a 16 bit *segment register*) and an `off` offset (hold by a 16 bit *general register*) while the corresponding physical address is simply computed using `seg*16 + off`: *as a result, no Global Descriptor Table, Local Descriptor Table, or paging table is needed by the CPU addressing circuit to translate a logical address into a physical one.* **Observer that in real mode no segment specific protection information are provided, therefore this modality is unsuitable for modern operating system**. Around 1MB (2^20 B) of memory is allowed

Protected Mode Similarly to real mode, a logical address consists of two parts: a **segment identifier** and an **offset** that specifies the relative address within the segment. However the target segment ***identifier*** is a **13-bit field** present into a **16 bit field** called the **segment selector** keep by 16 bit segment register (**remaining 3 bit are used for protection purposes**) while a 32 bit general register holds the offset. The corresponding physical address is computed using the linear address of the first byte of the segment (which is hold by a special table) using following equation: `TABLE[segment].base + offset`. In this modality Up to 4 GB of memory is allowed.

Long Mode Identical to protected mode but the offset is hold by 64-bit general registers (*although up to 48-bit are used in canonical form*). In this way is possible to address up to 2^{48} B (256 TB) of linear memory.

From now we describe protected mode. To make it easy to retrieve segment selectors quickly, the processor provides **segmentation registers** whose only purpose is to hold segment selectors; these registers are called `cs` (**code segment register, which points to a segment containing program instructions**), `ss` (**stack segment register, which points to a segment containing the current program stack**), `ds` (**data segment register, which points to a segment containing global and static data**), `es`, `fs`, and `gs` (**these three segmentation registers are general purpose and may refer to arbitrary data segments**.)

To be more precise, segment selectors have three fields:

index (13-bit) identifies the Segment Descriptor entry contained in the GDT or in the LDT (see below).

Table Indicator (1-bit) specifies whether the Segment Descriptor is included in the GDT (TI = 0) or in the LDT (TI = 1).

Requestor Privilege Level (RPL) (2-bit) specifies the **Current Privilege Leve**l of the CPU when the corresponding Segment Selector is loaded into the `cs` register. The value 0 denotes the highest privilege level, while the value 3 denotes the lowest one. Linux uses only levels 0 and 3, which are respectively called **Kernel Mode** and **User Mode.**

the 13-bit field which identifies the Segment Descriptor entry contained in the GDT or in the LDT (see below), 1-bit field which is te Table Indicator that specifies whether the Segment Descriptor is included in the GDT (TI = 0) or in the LDT (TI = 1). Requestor Privilege Level: specifies the Current Privilege Level of the CPU when the corresponding Seg- ment Selector is loaded into the cs register; it also may be used to selectively weaken the processor priv- ilege level when accessing data segments (see Intel documentation for details).

2.1.1 GDT and LDT

Each segment is represented by an **8-byte Segment Descriptor** that describes the segment characteristics. Segment Descriptors are stored either in the **Global Descriptor Table** (**GDT**) or in the **Local Descriptor Table** (**LDT**) **which are kept in main memory**.

The address (32 bit in protected mode, 64 bit in long mode) and size (always 16 bit) of the GDT in main memory are contained in the `gdtr` control register, while the address and size of the currently used LDT are contained in the `ldtr` control register.

GDT is used for the mapping of linear addresses *at least* for kernel mode (that is to manage kernel level segments) while **LDT is used for user mode** (each process is permitted to have its own LDT). **However GDT is the unique used segment table in most operating systems**.

2.1.2 Segment descriptor

A Segment Descriptor, that is an entry of the GDT/LDT, has **many** fields including:

Base Contains the linear address of the first byte of the segment.

G (Granularity flag) if equal to 0, the segment size is expressed in bytes; otherwise, it is expressed in multiples of 4096 bytes.

Limit Holds the offset of the last memory cell in the segment, *thus binding the segment length.* When G is set to 0, the size of a segment may vary between 1 byte and 1 MB; otherwise, it may vary between 4 KB and 4 GB.

DPL (Descriptor Privilege Level)(2 bit) : *used to restrict accesses to the segment.* It represents the minimal CPU privilege level requested for accessing the segment. Therefore, a segment with its DPL set to 0 is accessible only when the CPL is 0 (Kernel Mode) while a segment with its DPL set to 3 is accessible with every CPL value.

P (Segment-Present) is equal to 0 if the segment is not stored currently in main memory. Linux **always** sets this flag to 1, because it never swaps out whole segments to disk.

Now be careful. Because a Segment Descriptor is 8 bytes long, **its relative address inside the GDT/LDT is obtained by multiplying the 13-bit index field of the Segment Selector by 8**. For instance, if the GDT is at `0x00020000` (the value stored in the gdtr register) and the index specified by the Segment Selector is 2, the address of the corresponding Segment Descriptor is `0x00020000 + (2 x 8)`, or 0x00020010 (**all this in protected mode!!**)

The first entry of the GDT is always set to 0. This ensures that logical addresses with a `null` Segment Selector will be considered invalid, thus causing a processor exception.

There a sort of cache to speed up the access to segment descriptor. In fact for each of the six segmentation registers **there are additional non-programmable register which are used to contains the 8-byte Segment Descriptor**. Every time a Segment Selector is loaded in a segmentation register, the corresponding Segment Descriptor is loaded from memory into the matching nonprogrammable CPU register. From then on, translations of logical addresses referring to that segment can be performed without accessing the GDT.

After to have computed the address of a Segment Descriptor, to obtain the linear address, we adds the offset of the logical address to the Base field of the Segment Descriptor.

2.1.3 Segmentation in Linux

Linux uses segmentation in a very limited way. **All Linux processes running in User Mode use the same pair of segments to address instructions and data** which are called **user code segment** and **user data segment**, respectively. Similarly, **all Linux processes running in Kernel Mode use the same pair of segments to address instructions and data**: they are called **kernel code segment** and **kernel data segment**. Now there are some differences about values stored in the fields of their segmentation descriptors: for instance user data/code segments have a DPL equal to 3, while kernel data/code segments have a DPL equal to 0. All these segment descriptors have their base set to `0x00000000`.

The corresponding Segment Selectors are defined by the macros `__USER_CS`, `__USER_DS`, `__KERNEL_CS`, and `__KERNEL_DS`, respectively. To address the kernel code segment, for instance, the kernel just loads the value yielded by the `__KERNEL_CS` macro into the `cs` segmentation register. It is sufficient to load `__KERNEL_CS` into cs whenever the CPU switches to Kernel Mode (CPL equal to 0).

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

3.1 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.

3.2 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**.

3.3 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.

3.4 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

3.5 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`.

3.6 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 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.

3.7 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.

3.8 TLB operation

Besides general-purpose hardware caches, x86 processors include a cache called *Translation Lookaside Buffers* (**TLB**) to speed up linear address translation.

When a linear address is used for the first time, the corresponding physical address is computed through slow accesses to the Page Tables in RAM. The physical address is then stored in a TLB entry so that further references to the same linear address can be quickly translated.

In a multiprocessor system, **each CPU has its own TLB, called the local TLB of the CPU**. Contrary to the hardware cache, the corresponding entries of the TLB need **not** be synchronized, because processes running on the existing CPUs may associate the same linear address with different physical ones.

When the cr3 control register of a CPU is modified, the hardware automatically invalidates all entries of the local TLB, because a new set of page tables is in use (page table changes). However changes inside the current page table are not automatically reflected within the TLB.

Fortunately, Linux offers several TLB flush methods that should be applied appropriately, depending on the type of page table change:

flush_tlb_all This flushes the **entire TLB on all processors** running in the system, which makes it the most expensive TLB flush operation. It is used when we have made changes into the kernel page table entries. **After it completes, all modifications to the page tables will be visible globally to all processors.**

flush_tlb_mm(struct mm_struct *mm) Flushes all TLB entries of the non-global pages owned by a given process that is all entries related to the userspace portion for the requested mm context. Is used when forking a new process.

flush_tlb_range Flushes the TLB entries corresponding to a linear address interval of a given process and is used when releasing a linear address interval of a process (when mremap() or mprotect() is used).

flush_tlb_page Flushes the TLB of a single Page Table entry of a given process and is used when handling a page fault.

flush_tlb_pgtables Flushes the TLB entries of a given contiguous subset of page tables of a given process and is called when a region is being unmapped and the page directory entries are being reclaimed

Despite the rich set of TLB methods offered by the generic Linux kernel, every microprocessor usually offers a far more restricted set of TLB-invalidating assembly language instructions. **Intel microprocessors offers only two TLB-invalidating techniques: the automatic flush of all TLB entries when a value is loaded into the cr3 register and the invlpg assembly language instruction which invalidates a single TLB entry mapping a given linear address.**

The architecture-independent TLB-invalidating methods are extended quite simply to multiprocessor systems. **The function running on a CPU sends an Interprocessor Interrupt to the other CPUs that forces them to execute the proper TLB-invalidating function** (**expensive** operation (*direct cost*) due to latency for cross-CPU coordination in case of global TLB flushes).

Remember that flush a TLB has the direct cost of the latency of the firmware level protocol for TLB entries invalidation (selective vs non-selective). Recall that flush TLB lead to **indirect cost** of refilling TLB entries and the latency experimented by MMU firmware upon misses in the translation process of virtual to physical addresses.

3.8.1 When flush TLB?

As a general rule, **any process switch implies changing the set of active page tables and therefore local TLB entries relative to the old page tables must be flushed**; this is done automatically when the kernel writes the address of the new Page Global Directory into the cr3 control register.

Besides **process switches**, there are other cases in which the kernel needs to flush some entries in a TLB. For instance, when the kernel assigns a page frame to a User Mode process and stores its physical address into a Page Table entry, it must flush any local TLB entry that refers to the corresponding linear address (virtual addresses accessible **locally** in time-sharing concurrency). On multiprocessor systems, the kernel also must flush the same TLB entry on the CPUs that are using the same set of page tables, if any (virtual addresses accessible **globally** by every CPU/core in real-time-concurrency).

Kernel-page mapping has a *global* nature, therefore when we use vmalloc() / vfree() on a specific CPU, all the other must observer mapping updates and TLB flush is necessary.

4 Slide *'kernel-level-task-management'*

4.1 Interrupt handling

Under Linux, hardware interrupts are called **IRQ**'s (*Interrupt Requests*) and their management typically occurs via a **two-level logic**:

Top Half A routine that actually responds to the interrupt and do a minimal amount of work to schedule its bottom half (this operation is very fast).

Bottom Half A routine scheduled by top half which execute whatever other work is required to handle the interrupt (such as awakening processes, starting up another I/O operation, and so on)

For instance, when a network interface reports the arrival of a new packet, the top half routine just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.

The most important aspect of this setup it that it permits the *top half to service a new interrupt while the bottom half is still working*; **in fact all interrupts are enabled during execution of the bottom half**. Generally the execution of top half code is handled according to a *non-interruptible scheme* (**but isn't mandatory**).

This scheme permit to **avoid to keep locked resources when an interrupt occurs** (we may incur the risk of delaying critical actions as a spin-lock release) **avoiding possible deadlocks** when a slow interrupt management is hit by the activation of another one that needs the same resources. **Moreover this scheme keep kernel response time small which is a very important property for many time-critical applications that expect their interrupt requests to be serviced in a few milliseconds.**

4.2 Softirqs, Tasklets and work queues

Form Linux 2.6, two different mechanisms are used to implement top/bottom-half processing:

- The so-called *deferrable functions*, which we will call as **softirqs** and **tasklets**: they are very fast, but all tasklet code must be atomic.
- The **Workqueues**, which may have a higher latency but that are allowed to sleep.

4.2.1 Softirqs

Softirqs are statically allocated, that is they are defined at compile time. The main data structure used to represent softirqs is the `softirq_vec` array, which includes `NR_SOFTIRQS` (32 entries) elements of type `softirq_action`. **Observer that the priority of a softirq is the index of the corresponding `softirq_action` element inside the array**. Some of the softirqs used in Linux are:

`HI_SOFTIRQ` With priority equal to 0 (first element of array) and it handles high priority tasklets.

`TIMER_SOFTIRQ` With priority equal to 1 and it is used for timer related interrupts.

Another crucial data structure for implementing the softirqs is a **per-CPU 32-bit mask describing the pending softirqs**; it is stored in the `__softirq_pending` field of the `irq_cpustat_t` data structure (which is one of the data structure used per each CPU in the system). To get and set the value of the bit mask, the kernel makes use of the `local_softirq_pending()`. This is way softirqs can run concurrently on several CPUs, even if they are of the same type.

During interrupt acceptance, top half routine set properly the bit mask in the `__softirq_pending` field and then exit.

Checks for active (pending) softirqs should be perfomed periodically, but without inducing too much overhead. They are performed in a few points of the kernel code.

For this purpose, Linux, **for each CPU**, uses the so called `ksoftirqd/n` kernel thread (where n is the logical number of the CPU) to manage softirqs array executing bottom halves asynchronously. Once awaken, that thread, running the `ksoftirqd()` function, checks softirq bit mask for pending softirqs inspecting the per-CPU field `__softirq_pending`. If there are no softirqs pending, the function puts the current thread in the `TASK_INTERRUPTIBLE` state and invokes then the `cond_resched()` function to perform a process switch; otherwise, the thread runs the softIRQ handler, running `do_softirq()`.

Be careful that the top half routine can set the bit mask telling that a `ksoftirqd/x` awaken on a CPU-core x will not process the handler associated with a given softIRQ; in this way we can **create affinity between SoftIRQs and CPU-cores in order to exploit NUMA machines**. Is also possible to set bit mask in order to build affinity on group of CPU for load balancing; **in other word is possible a multithread execution of bottom half tasks**.

4.2.2 tasklet

When we use softirqs not necessarily we queue bottom half task, so this setup can be even more responsive. However the queuing concept is still there for on demand usage, if required.

Tasklets are built on top of two softirqs named `HI_SOFTIRQ` and `TASKLET_SOFTIRQ`. Several tasklets may be associated with the same softirq, each tasklet carrying its own function. There is no real difference between the two softirqs, except that `do_softirq()` executes `HI_SOFTIRQ`'s tasklets before `TASKLET_SOFTIRQ`'s tasklets.

Tasklets and high-priority tasklets are stored in the `tasklet_vec` and `tasklet_hi_vec` arrays respectively and both of them include `NR_CPUS` elements which are **list** of **tasklet descriptors** which are a data structure of type `tasklet_struct`

Each `tasklet_struct` has many fields including the `state` field which represents the status of current tasklet and can assume two value: `TASKLET_STATE_SCHED` (tasklet pending), `TASKLET_STATE_RUN` (tasklet is running). Using this field is possible to keep track of a specific bottom half task, related to the execution of a specific function internal to the kernel.

Linux offers many APIs to manage tasklets: for instance to allocate a new `tasklet_struct` data structure and initialize is need to invoke `tasklet_init()`; this function receives as its parameters the address of the tasklet descriptor , the address of your tasklet function (`void (*func)`), and its optional integer argument (`unsigned long`) for data.

The tasklet may be selectively disabled by invoking either `tasklet_disable_nosync()` or `tasklet_disable()`. Both functions increase the count field of the tasklet descriptor, but the latter function does not return until an already running instance of the tasklet function has terminated. To reenable the tasklet, use `tasklet_enable()`. To activate the tasklet, you should invoke either the `tasklet_schedule()` function or the `tasklet_hi_schedule()` function, according to the priority that you require for the tasklet. When a tasklet is enabled its descriptor is added at the beginning of the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]`, where n denotes the logical number of the local CPU; then `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` softirq are enabled (**all these operation are executed with local interrupts disabled**)

Remember that tasklets can be instantiated by exploiting also the following macros defined in include `include/linux/interrupt.h`:

- `DECLARE_TASKLET(tasklet, function, data)`
- `DECLARE_TASKLET_DISABLED(tasklet, function, data)`

Finally to execute tasklet associated with the `HI_SOFTIRQ` softirq we run `tasklet_hi_action()`, while for those associated with `TASKLET_SOFTIRQ` we use `tasklet_action()`.

Observer that if the tasklet has already been scheduled on a different CPU-core, it will not be moved to another CPU-core if it's still pending (generic softirqs can instead be processed by different CPU-cores)

Tasklets run in interrupt context (see below)

4.2.3 Work queue

The work queues have been introduced in Linux 2.6 and replace a similar construct called "task queue" used in Linux 2.4. Also the work queues are used to allow kernel functions to be activated and later executed by special kernel threads called *worker threads*.

However there is one important difference with softirq and tasklet: **deferrable functions (that is softirqs and tasklet) run in interrupt context while functions in work queues run in process context**. *Running in process context is the only way to execute functions that can block (for instance, functions that need to access some block of data on disk) because no process switch can take place in interrupt context.*

Observer that interrupts are enabled while the work queues are being run (except if the same work to be done disables them)

The main data structure associated with a work queue is a descriptor called `workqueue_struct`, which contains **many** fields including an array of `NR_CPUS` elements (the maximum number of CPUs in the system.) Each element is a descriptor of type `cpu_workqueue_struct`, which contains a `worklist` field which is the head of a doubly linked list collecting the pending functions of the work queue. Every pending function is represented by a `work_struct` data structure.

The `create_workqueue("foo")` function receives as its parameter a string of characters and returns the address of a `workqueue_struc` descriptor. The function also creates n worker threads (where n is the number of CPUs effectively present in the system), named after the string passed to the function: `foo/0, foo/1`, and so on. The `create_singlethread_workqueue()` function is similar, but it creates just one worker thread, no matter what the number of CPUs in the system is. To destroy a work queue the kernel invokes the `destroy_workqueue()` function, which receives as its parameter a pointer to a `workqueue_struct` array.

Another very important API is `queue_work()` which inserts a function (already packaged inside a `work_struct` descriptor) in a work queue; it receives a pointer `wq` to the `workqueue_struct` descriptor and a pointer work to the `work_struct` descriptor.

The `queue_delayed_work()` function is nearly identical to `queue_work()`, except that it receives a third parameter representing a time delay in system ticks and it is used to ensure a minimum delay before the execution of the pending function.

`cancel_delayed_work()` cancels a previously scheduled work queue function. The `flush_workqueue()` function receives a `workqueue_struct` descriptor address and blocks the calling process until all functions that are pending in the work queue terminate.

4.2.4 The predefined work queue

The kernel offers a predefined work queue called *events*, which can be freely used by every kernel developer. The predefined work queue is nothing more than a standard work queue that may include functions of different kernel layers and I/O drivers.

To make use of the predefined work queue, the kernel offers some APIs including `schedule_work(struct work_struct *work)` and `schedule_work_on(int cpu, struct work_struct *work)`.

4.3 `container_of`

The macro `container_of(ptr, type, member)` takes, as you can see, three arguments: a pointer to the member of a data structure, the name of the type of the data structure, and the name of the member the pointer refers to. The macro yields the address of the container structure which accommodates the specified member.

4.4 Timers

On the x86 architecture, the kernel must explicitly interact with several kinds of clock circuits which are used both to keep track of the current time of day and to make precise time measurements. **The timer circuits are programmed by the kernel, so that they issue interrupts at a fixed, predefined frequency; such periodic interrupts are crucial for implementing the software timers used by the kernel and the user programs**.

Time Stamp Counter (TSC) It is a counter accessible through the 64-bit *Time Stamp Counter* (**TSC**) register, which can be read using `rdtsc` assembly language instruction. **It represents a counter that is increased at each clock signal.** It is used by Linux to determine the clock signal frequency while initializing the system; that task is accomplished using `calibrate_tsc()`.

High Precision Event Timer (HPET) The HPET represents a very powerful chip which provides up to **eight 32-bit or 64-bit independent counters exploitable by kernel**. Each counter is driven by its own clock signal, whose frequency must be at least 10 MHz and, therefore, the counter is increased at least once in **100 nanoseconds**. Any counter is associated with at most 32 timers, each of which is composed by a *comparator* and a *match register*. **The comparator is a circuit that checks the value in the counter against the value in the match register, and raises a hardware interrupt if a match is found. Some of the timers can be enabled to generate a periodic interrupt.**

LAPIC The Local APIC Timer (LAPIC-T) represents another time-measuring device. This timer has a counter of **32 bits long** used to store the number of of ticks that must elapse before the interrupt is issued; therefore, the local timer can be programmed to issue interrupts at very low frequencies. **Observe that local APIC timer sends an interrupt only to its processor**. The APIC's timer is based on the bus clock signal and can be can be programmed in such a way to decrease the timer counter every 1, 2, 4, 8, 16, 32, 64, or 128 bus clock signals.

4.4.1 The timer interrupt handler

As said these timer circuits issues special interrupts called **timer interrupt**, which notifies the kernel that one more time interval has elapsed. Interrupts can both involve a specific CPU (CPU local timer interrupt signals timekeeping activities related to the local CPU, such as monitoring how long the current process has been running and updating the resource usage statistics) or signal activities not related to a specific CPU, such as handling of software timers and keeping the system time up-to-date.

4.5 Preemption

As a general definition a kernel is *preemptive* **if a process switch may occur while the replaced process is executing a kernel function, that is, while it runs in Kernel Mode**.

Kernel pre-emption is disabled when the `preempt_count` field in the `thread_info` descriptor referenced by the `current_thread_info()` macro is greater than zero. Linux provides several APIs to manage kernel pre-emption:

`preempt_count()` Return the `preempt_count` field in the `thread_info` descriptor.

`preempt_disable()` Increases by one the value of the preemption counter.

`preempt_enable_no_resched()` Decreases by one the value of the preemption counter.

`preempt_enable()` Decreases by one the value of the preemption counter, and invokes `preempt_schedule()` if the `TIF_NEED_RESCHED` flag in the `thread_info` descriptor is set

But there are other two very important API and they are related to **per-CPU variables**.

Remember that a ***per-CPU variables* is an array of data structures, one element per each CPU in the system**. A CPU should not access the elements of the array corresponding to the other CPUs; on the other hand, it can freely read and modify its own element without fear of race conditions, because it is the only CPU entitled to do so. **While per-CPU variables provide**

protection against concurrent accesses from several CPUs, they do **not provide protection against accesses from asynchronous functions** (interrupt handlers and deferrable functions like tasklet and softirqs). **Therefore per-CPU variables are prone to race conditions caused by kernel pre-emption, both in uniprocessor and multiprocessor systems**. **As a general rule, a kernel control path should access a per-CPU variable with kernel preemption disabled.**. We have some API:

`get_cpu_var(name)` Disables kernel preemption, then selects the local CPU's element of the per-CPU array name

`put_cpu_var(name)` Enables kernel preemption.