Contents

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*

According to Tomasulo's scheme, register renaming is provided by special buffers called **reservation stations**, which have three fields:

OP the operations to be executed.

Q1, Q2 the reservation stations that will produce the input for OP.

V1, V2 the values of the source operands.

In other words, **reservation station are used to hold the operands of instructions, fetching and buffering an operand as soon as it is available, eliminating the need to get the operand from a register**. Results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with the **common data bus** (**CDB**) that allows all units waiting for an operand to be loaded simultaneously (*In pipelines with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed*).

If one or more of the operands is not yet available the common data bus will be monitored. When an operand becomes available, it is placed into any reservation station awaiting it. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available, RAW hazards are avoided.

When successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming. **Each functional unit has a single reservation station. The functional unit begins processing when it is free and when all source operands needed for an instruction are real.**

A **re-order buffer** (ROB) is used TO acquire all the newly produced instruction results and keeps them uncommitted up to the point where the instruction can be committed in-order.

1.3 Dynamic Branch Prediction

The hardware support for reducing the performance losses of branches is able to predict how they will behave. The behaviour of branches can be predicted both **statically** (*to predict a branch as taken always: not very accurate*) at compile time and **dynamically** by the hardware at execution time, make predictions on the basis of profile information collected from earlier runs.

The simplest dynamic branch-prediction scheme is a **branch-prediction buffer** or **branch history table**.

It is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. The (speculative) execution flow follows the direction related to the prediction by the status bit, thus following the recent behaviour since recent past is expected to be representative of near future.

However this simple 1-bit prediction scheme has a performance shortcoming: **Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the mis-prediction causes the prediction bit to be flipped**.

To remedy this weakness, 2-bit prediction schemes are often used. **In a 2-bit scheme, a prediction must miss twice before it is changed** (that is require 2 subsequent prediction errors for inverting the prediction). Benchmarks on 2-bit branch-prediction buffer with 4096 entries results in a prediction accuracy ranging from over 99% to 82%, or a mis-prediction rate of 1% to 18%.

However this predictor schemes use only the recent behaviour of a single branch to predict the future behaviour of that branch. **It may be possible to improve the prediction accuracy if we also look at the recent behaviour of other branches rather than just the branch we are trying to predict.**

Branch predictors that use the behaviour of other branches to make a prediction are called **correlating predictors** or **two-level predictors**. In the general case an (m, n) predictor uses the behaviour of the last m branches; that is there are 2^m branch predictors, each of which is an n-bit predictor for a single branch. The global history of the most recent m branches can be recorded in m-bit shift register, where each bit records whether the branch was taken or not taken. The branch-prediction buffer can then be indexed using a concatenation of the low-order bits from the branch address with the m-bit global history.

A 2-bit predictor with no global history is simply a (0,2) predictor.

Tournament predictors makes prediction using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global, or even some mix) was most effective in recent predictions. The advantage of a tournament predictor is its ability to select the right predictor for a particular branch.

Indirect branches are for which the target is not know at instruction fetch time. A table of 2-bit predictors are used which is indexed using lower portion of the address of the branch instruction.

Loop unrolling is a software technique that allows reducing the frequency of branches when running loops, and the relative cost of branch control instructions. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code, allowing from different iterations to be scheduled together reducing the cost of branch.

Unrolling improves the performance of loop by eliminating overhead instructions, although it increases code size substantially (consequently locality and cache efficiency may degrade significantly) and the use of a more registers (leading to more frequent memory interactions).

1.4 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.

1.5 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.

1.6 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.**

1.7 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. Linux prefers paging to segmentation for the following reasons:

1. Memory management is simpler when all processes use the same segment register values, that is, when they share the same set of linear addresses.

2. **One of the design objectives of Linux is portability to a wide range of architectures** (RISC architectures in particular have limited support for segmentation).

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).

2.1.4 GDT

There is one GDT for every CPU in the system that is we have some replicated GDT which differs by some entries (this is important for performance motivation and for transparency of data access separation (same segment = to access different linear addresses by more CPU)). All GDTs are stored in the `cpu_gdt_table` array. **Each GDT includes 18 segment descriptors and 14 unused, or reserved entries including the null one**. The most important segment descriptors are:

- Four segment descriptors corresponding to user and kernel code and data segments (see previous section).
- A **Task State Segment** (TSS), different for each processor in the system.
- A segment including the default Local Descriptor Table (LDT).
- Three **Thread-Local Storage** (TLS) segments: **this is a mechanism that allows multithreaded applications to make use of up to three segments containing data local to each thread**. The `set_thread_area()` and `get_thread_area()` system calls, respectively, create and release a TLS segment for the executing process.

2.2 TSS

The 80x86 architecture includes a specific segment type called the **Task State Segment** (**TSS**). Linux uses the tss_struct structure to describe the format of the TSS. As already mentioned in **The Task State Segments are sequentially stored into `init_tss` array which stores one TSS for each CPU on the system**: in particular, the Base field of the TSS descriptor for the nth CPU points to the nth component of the `init_tss` array. The DPL is set to 0, because processes in User Mode are not allowed to access TSS segments.

According to the original Intel design, TSS is normally used to store hardware contexts in case of process switch. However, **Linux doesn't use hardware context switches because it uses a single TSS for each processor, instead of one for every process**. Linux use the TSS in only two cases:

- when the CPU switches from User Mode to Kernel Mode, **it fetches the address of the Kernel Mode stack from the TSS**

  Note that each process descriptor includes a field called `thread` of type `thread_struct`, in which the kernel saves the hardware context. This data structure includes fields for most of the CPU registers, except the general-purpose registers such as eax, ebx, etc., which are stored in the Kernel Mode stack.

- When a User Mode process attempts to access an I/O port, the CPU may need to access an I/O Permission Bitmap stored in the TSS to verify whether the process is allowed to address the port.

80x86 architecture provides a segment register called the task register (`TR`) to hold a segment selector that points to a valid TSS segment descriptor which resides in the GDT. To initialize TR, 80x86 ISA provides the assembly instruction `ltr` which loads the source operand into the task register, where the source operand (a general-purpose register or a memory location) contains a segment selector that points to the TSS contained into GDT. After that, the CPU uses the segment selector into `tr` to locate the segment descriptor for the TSS in GDT.

2.3 Control register

A **control registers** are a set of registers which changes or controls the general behavior of a CPU like its addressing mode, paging control, and coprocessor control and so on. In modern x86 machines we have:

CR0 On x86-64 processors in long mode, it is 64 bits long register which have many control bits. The most important are:

- Bit 0 - PE Protected Mode Enable -If 1, system is in protected mode, else system is in real mode.
- Bit 31 - PG Paging If 1, enable paging and use the CR3 register, else disable paging.
- Bit 16 - WP Write protect When set, the CPU can't write to read-only pages when privilege level is 0

CR1 Reserved.

CR2 Contains a value called *Page Fault Linear Address* (**PFLA**). **When a page fault occurs, the address the program attempted to access is stored in the CR2 register.**

CR3 Used when virtual addressing is enabled, hence when the PG bit is set in CR0. CR3 holds a pointer to the *page directory* for the current task

2.4 Management of interrupts

Linux keeps a system table called **Interrupt Descriptor Table** (**IDT**) to **associate to each interrupt or exception vector with the address of the corresponding handler**.

On each CPU, the `idtr` (*interrupt descriptor table register*) CPU register allows the IDT to be located in memory. In fact, in protected mode it holds:

1. a 48 bit field (*64 bit in long mode*) which specifies the IDT base linear address.
2. a 16 bit field which provides the IDT max length, that is the number of entries currently present in it.

There are two assembly instruction capable to manipulate the IDT:

`lidt` **(Load IDT)** It is used to Load the values in the source operand into IDT.

`sidt` **(Store IDT)** It is used to store the content the interrupt descriptor table register in the destination operand.

Remember that the IDT must be initialized by Linux before enabling interrupts by using the `lidt` assembly language instruction. IDT is initialized with an excep- tion handler function for each recognized exception by `trap_init()` function (`/usr/src/linux/kernel/traps.c`). For example the interrupt 0x80 is associated to `_system_call` entry point using `set_system_gate (0x80, &system_call)`.

In protected mode, each IDT's entry is an 8-byte (64 bit) descriptor of type struct `desc_struct` (which is defined in `include/asm-i386/desc.h`). In long mode each entry in the IDT grows by 64-bits. The IDT contains some descriptor's type and the value of the `Type` field encoded in the bits 40–43 identifies the descriptor type:

Interrupt Gate Descriptor They are used to handle interrupts. *Includes the Segment Selector and the offset inside the segment of an interrupt or exception handler.* While transferring control to the proper segment, the processor clears the `IF` (**Interrupt flag**) to disable further maskable interrupts (bit 40 set to 0, 41-43 to 1).

Trap Gate Descriptor They are used to handle exceptions. Similar to an interrupt gate, except that while transferring control to the proper segment, the processor does not modify the IF flag (bit 40 set to 1, 41-43 to 1).

A fully populated IDT is 2 KB (256 entries of 8 bytes each) in length. **All entry from 0 to 31 are reserved by Intel for processor generated exceptions** (general protection fault, page fault, etc.) while all the other entries are available for system programming purposes.

The following architecture-dependent functions (defined in `arch/i386/kernel/traps.c`) are used to insert on entry in the IDT (note that `n` indicates the target entry of the IDT, while `addr` indicates the address of the software module to be invoked for handling the trap or the interrupt):

`set_trap_gate(n,addr)` Inserts a trap gate in the nth IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to addr. **The DPL field is set to 0** (*we cannot rely on the `int` assembly instruction unless we are already executing in kernel mode*).

`set_intr_gate(n,addr)` Inserts an interrupt gate in the nth IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to addr. The DPL field is set to 0.

`set_system_gate(n,addr)` Inserts a trap gate in the nth IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to addr. **The DPL field is set to 3**.

2.4.1 System calls

Linux System Call Table

The following table lists the system calls for the Linux 2.2 kernel. It could also be thought of as an API for the interface between user space and kernel space. My motivation for making this table was to make programming in assembly language easier when using only system calls and not the C library (for more information on this topic, go to http://www.linuxassembly.org). On the left are the numbers of the system calls. This number will be put in register

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

3.1 Pagination

The paging unit thinks of all RAM as partitioned into fixed-length *page frames*. Each page frame contains a *page*, that is, the length of a page frame coincides with that of a page. A page frame is a constituent of main memory, and hence it is a storage area. It is important to distinguish a page from a page frame; the former is just a block of data, which may be stored in any page frame or on disk.

The data structures that map linear to physical addresses are called **page tables**; *they are stored in main memory and must be properly initialized by the kernel before enabling the paging unit.*

In 32-architectures, the linear address is 32 bit long and is divided into three fields:

Directory The most significant 10 bits.

Table The intermediate 10 bits.

Offset The least significant 12 bits.

The physical address of the Page Directory in use is stored in a control register named `cr3`. The translation of linear addresses is accomplished in this way:

1. The *Directory field* within the linear address determines the entry in the Page Directory that points to the proper Page Table.
2. The *Table field*, in turn, determines the entry in the Page Table that contains the physical address of the page frame containing the page.
3. The Offset field determines the relative position within the page frame.

Because Offset field is 12 bits long, each page consists of 4096 bytes of data; therefore paging unit of Intel processors handles 4 KB pages.

Both the Directory and the Table fields are 10 bits long, so Page Directories and Page Tables can include up to 1024 entries. It follows that a Page Directory can address up to $1024 \times 1024 \times 4096 = 2^{32}$ memory cells.

Up to version 2.6.10, the Linux paging model consisted of three paging levels, but, starting with version 2.6.11, a four-level paging model has been adopted.

The four types of page tables are called:

Page Global Directory Its entry type is `pgd_t`

Page Upper Directory Its entry type is `pud_t`

Page Middle Directory Its entry type is `pmd_t`

Page Table Its entry type is `pte_t`

The Page Global Directory includes the addresses of several Page Upper Directories, which in turn include the addresses of several Page Middle Directories, which in turn include the addresses of several Page Tables. Each Page Table entry points to a page frame. **Entry data types (defined into `include/asm-i386/page.h`) are 64-bit data types when PAE is enabled and 32-bit data types otherwise.**

Linux adopts some trick in order that paging model fits both 32-bit and 64-bit architectures maintaining the same code and structure:

32-bit architectures (PAE not enabled) Since two paging levels are sufficient, Linux kernel set the number of entries of the Page Upper Directory and Page Middle Directory to 1.

32-bit architectures (PAE enabled) Three paging levels are used; in this case the Page Upper Directory is virtually eliminated setting its size to 1.

The following macros define the size of the page tables blocks (`include/asm-i386/pgtable-2level.h`):

- `#define PTRS_PER_PGD 1024`
- `#define PTRS_PER_PMD 1`
- `#define PTRS_PER_PTE 1024`

3.1.1 Kernel Page Initialization

We now describe how the kernel initializes its own page tables. **Right after the kernel image is loaded into memory, the CPU is still running in real mode; thus, paging is not enabled.**

First of all, all normal code in `vmlinuz` is compiled with the base address at `PAGE_OFFSET` + 1 MB; **therefore the kernel is actually loaded starting from the first megabyte of memory** (the first megabyte is typically used by BIOS to store system hardware configuration after `Power-On Self-Test` (POST) and for some BIOS routines used to interact with some devices; therefore, this memory area is ignored by Linux).

A provisional Page Global Directory is initialized statically during kernel compilation and it is contained into the `swapper_pg_dir` variable (now `init_level4_pgt` on x86-64/kernel3 or `init_top_pgt` on x86-64/kernel4)), placed using linker directives at 0x00101000, while the provisional Page Tables are initialized by the `startup_32()` assembly language function defined in `arch/i386/kernel/head.S`.

In the first phase, the kernel creates, invoking `alloc_bootmem_low_pages(number of page)` (defined in `include/linux/bootmem.h`) a limited address space including the kernel's code and data segments, the initial Page Tables, and 128 KB for some dynamic data structures: all of this fit in the first 8 MB of RAM. The objective of this phase of paging is to allow these 8 MB of RAM to be easily addressed.

The Kernel creates the desired mapping by filling all the `SWAPPER_PG_DIR` entries with zeroes, except for entries 0, 1, 0x300 (decimal 768), and 0x301 (decimal 769); The address field of entries 0 and 0x300 is set to the physical address of pg0, while the address field of entries 1 and 0x301 is set to the physical address of the page frame following pg0.

3.1.2 Describing a Page Table Entry

As mentioned, each entry is described by the structs `pte_t`, `pmd_t`, `pud_t` and `pgd_t`. Even though these are often just `unsigned integers`, they are defined as structs for type protection so that they will not be used inappropriately.

For type casting, several macros are provided in `asm/page.h`, which takes the above types and returns an unsigned integer. They are `pte_val()`, `pmd_val()`, `pud_val()` e `pgd_val()`. To cast an unsigned integer into the required type other more macros are provided `__pte()`, `__pmd()`, `__pgd()` and `__pud`.

The entries of Page Directories and Page Tables have the same structure. Each entry includes many filed. the most important are:

Field containing the 20 most significant bits of a page frame physical address Field containing the 20 most significant bits of a page frame physical address. Because each page frame has a 4-KB capacity, its physical address must be a multiple of 4096, so the 12 least significant bits of the physical address are always equal to 0. If the field refers to a Page Directory, the page frame contains a Page Table; if it refers to a Page Table, the page frame contains a page of data.

Present flag If it is set, the referred-to page (or Page Table) is contained in main memory; if the flag is 0, the page is not contained in main memory.

Access flag Set each time the paging unit addresses the corresponding page frame. This flag may be used by the operating system when selecting pages to be swapped out. The paging unit never resets this flag; this must be done by the operating system.

Dirty flag Applies only to the Page Table entries. It is set each time a write operation is performed on the page frame. As with the Accessed flag, Dirty may be used by the operating system when selecting pages to be swapped out. The paging unit never resets this flag; this must be done by the operating system.

Read/Write flag Contains the access right (Read/Write or Read) of the page or of the Page Table. (Zero means read only access)

User/Supervisor flag Contains the privilege level required to access the page or Page Table (Zero means supervisor privilege)

Global flag Applies only to Page Table entries. This flag was introduced to prevent frequently used pages from being flushed from the TLB cache. It works only if the Page Global Enable (PGE) flag of register `cr4` is set. Non-zero means that the corresponding TLB entry will not be flushed upon loading a new value into the page table pointer `cr3`.

Page Size flag Applies only to Page Directory entries. If it is set, the entry refers to 4 MB page frame.

Present flag If it is set, the referred-to page (or Page Table) is contained in main memory; if the flag is 0, the page is not contain

---

There are **many** macro in `include/asm-i386/pgtable.h` to set or read above fields within the entries of PTE or PGD like `pte_exec()` (Reads the User/Supervisor flag) or `pte_mkread()` (Sets the User/Supervisor flag) (all macros are architecture dependent). There are also macros for masking and setting those above fields like `#define _PAGE_PRESENT 0x001` or `#define _PAGE_RW 0x002` ecc.

Recall that upon a TLB miss, firmware accesses the page table and then, after pick an entry, it first checked present flag: **if that flag is set to zero, a page fault occurs which gives rise to a trap.** After the execution of page fault handler, the instruction that gave rise to the trap can get finally re-executed but rememeber that it might rise additional traps (for instance, when it attempts to access a read only page in write mode (rising a segmentation fault))

3.2 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.

3.3 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**.

3.4 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.

3.5 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

3.6 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`.

3.7 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.

3.8 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.

3.9 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.

3.9.1 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 TCB

In Linux, the process descriptor is represented by a `task_struct` structures (defined in `include/linux/sched.h`) which have many fields including:

`volatile long state` Process state.

`struct mm_struct *mm` Pointer to memory are descriptor.

`thread_info` Low-level information (CPU specific) for the process.

`files_struct` Pointers to file descriptors.

`signal_struct` Signals received.

`pid_t pid`

`volatile long need_resched`

`long nice`

Each `task_struct` structure includes a tasks field of type `list_head` (*a doubly linked list defined by Linux*) whose `prev` and `next` fields point, respectively, to the previous and to the next `task_struct` element.

The head of the process list is the `init_task` descriptor (type `task_struct`); it is the process descriptor of the so-called process 0 or *swapper* or *IDLE process*.

The `state` field consists of an array of flags, each of which describes a possible process state. In the current Linux version, these states are mutually exclusive, and hence exactly one flag of state always is set; the remaining flags are cleared. The possible states (there are some macro defined in `include/linux/sched.h`) are:

`TASK_RUNNING` The process is either executing on a CPU or waiting to be executed.

`EXIT_ZOMBIE` Process execution is terminated, but the parent process has not yet issued wait() system call to return information about the dead process.

`TASK_INTERRUPTIBLE` The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt is an example of conditions that might wake up the process

When looking for a new process to run on a CPU, the kernel has to consider only the runnable processes (that is, the processes in the `TASK_RUNNING` state).

4.5.1 TCB allocation

Processes are dynamic entities, whose lifetimes range from a few milliseconds to months, **therefore process descriptors are stored in dynamic memory rather than in the memory area permanently assigned to the kernel**. For each process, **Linux packs two different data structures in a single per-process memory area**:

- a small data structure linked to the process descriptor, namely the `thread_info` structure.
- Kernel Mode process stack and the length of this memory area is usually 8,192 bytes (8 KB) (**stored in two consecutive page frames with the first page frame aligned to a multiple of** 2^13).

The `thread_info` structure resides at the beginning of the memory area, and the stack grows downward from the end. The growth of the stack size may lead to buffer overflow risks, if the stack size is not rescaled and memory fragmentation, if the stack size is rescaled

The C language allows the `thread_info` structure and the kernel stack of a process to be conveniently represented by means of the following union construct:

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[2048]; /* 1024 for 4KB stacks */
};
```

4.6 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.

4.6.1 Process schedule

Linux processes are preemptable and therefore, when a process enters in the `TASK_RUNNING` state, **the kernel checks whether its *dynamic priority* is greater than the priority of the currently running process**. If it is, the execution of current process is interrupted and the scheduler is invoked to select another process to run. **A process also may be pre-empted when its time quantum expires**, that is when the `TIF_NEED_RESCHED` flag in the `thread_info` structure of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.

The scheduler always succeeds in finding a process to be executed; in fact, there is always at least one runnable process, that is the process 0, also known as *swapper* or *idle process* (**every CPU of a multiprocessor system has its own swapper process with PID equal to 0**).

Every Linux process is always scheduled according to one of the following scheduling classes:

`SCHED_FIFO` A First-In, First-Out

`SCHED_RR` A Round Robin.

`SCHED_NORMAL` Time-shared.

4.6.2 Process Priority

Every conventional (*non-real-time*) process has its own **static priority** which is used to determine the *base time quantum* of a process, which is computed so that an higher static priority corresponds to a longer base time quantum.

The static priority is represented by a number ranging from 100 (highest priority) to 139 (lowest priority). A new process always inherits the static priority of its parent, although an user can change the static priority of the processes that he owns using `nice()` or `setpriority()` system call.

However every conventional process also has a **dynamic priority**, which is a value ranging from 100 (highest priority) to 139 (lowest priority). **The dynamic priority is the number actually looked up by the scheduler when selecting the new process to run: a thread that calls the schedule function can be pre-empted by one that has higher dynamic priority**.

It is related to static priority in term of reward or penalty depending on whether the thread is interactive or not.

Generally a thread is considered interactive if its sleep time is very high; technically a process having highest static priority (100) is considered interactive when its average sleep time exceeds 200 ms.

Obliviously all static and dynamic priority values of a process is stored into its TCB.

4.6.3 Data structures used by the scheduler

The `runqueue` data structure is the most important data structure of the Linux 2.6 scheduler. It links the process descriptors of all runnable processes, that is of those in a `TASK_RUNNING` state. **Each CPU in the system has its own `runqueue` and all `runqueue` structures are stored in the `runqueues` per-CPU variable**. *However a CPU-core can access to the runqueue of another one for load balancing purposes*

The most important fields of the `runqueue` data structure are those related to the lists of runnable processes. They are:

Type	Field	Description
`prio_array_t *`	`active`	Pointer to the lists of active processes. Technically it points to one of the two `prio_array_t` data structures in `arrays`
`prio_array_t *`	`expired`	Pointer to the lists of expired processes. Technically it points to one of the two `prio_array_t` data structures in `arrays`
`prio_array_t [2]`	`arrays`	The two sets of active and expired processes.

Every runnable process in the system belongs to one, and just one, runqueue. As long as a runnable process remains in the same runqueue, it can be executed only by the CPU owning that runqueue. **Remember that runnable processes may migrate from one runqueue to another.** *As you see, there is no mix of runnable and non-runnable tasks on a runqueue.*

As you can see, the `arrays` field of the `runqueue` is an array consisting of two `prio_array_t` structures. **Each data structure represents a set of runnable processes, and includes 140 doubly linked list heads (one list for each possible process priority), a priority bitmap, and a counter of the processes included in the set**. *Remember that 40 levels (say [100-139]) map to classical Unix time-sharing, while 100 levels (say [0-99]) map to Unix real-time scheduler extensions.*

Periodically, the role of the two data structures in arrays changes: the active processes suddenly become the expired processes, and the expired processes become the active ones. We switch the queues upon a new epoch.

When a process creates a child, **the number of ticks left to the parent is split in two halves**: one for the parent and one for the child. *This is done to prevent users from getting an unlimited amount of CPU time under certain condition.*

5 Slide *'trap-interrupt-architecture'*

5.0.1 IPI

An **Inter-processor Interrupt** (**IPI**) **allow a CPU to send interrupt signals to any other CPU in the system**. An interprocessor interrupt is delivered directly as a message on the bus that connects the **local APIC** (*Advanced Programmable Interrupt Controller*) of all CPUs. **An IPI is a synchronous event from the sender CPU-core point of view while it is an asynchronous one from recipient CPU-core point of view.** Classically, at least two priority levels are admitted:

High Leads to immediate processing of the IPI at the recipient (*a single IPI is accepted and stands out at any point in time*)

Low Low priority generally leads to queue the requests and process them via sequentialization

Each local APIC of each CPU of the system has several 32-bit registers (which can be used for posting IPI requests in the system), an internal clock; a local timer device; and two additional IRQ lines, `LINT0` and `LINT1`, reserved for local APIC interrupts. All local APICs are connected to an external I/O APIC, giving rise to a multi-APIC system.

The I/O APIC consists of a set of 24 IRQ lines, a 24-entry Interrupt Redirection Table, programmable registers, and a message unit for sending and receiving APIC messages over the APIC bus.

The multi-APIC system allows CPUs to generate interprocessor interrupts. When a CPU wishes to send an interrupt to another CPU, it stores the interrupt vector and the identifier of the target's local APIC in the **Interrupt Command Register** (**ICR**) of its own local APIC. A message is then sent via the APIC bus to the target's local APIC, which therefore issues a corresponding interrupt to its own CPU.

Interrupt requests coming from external hardware devices can be distributed among the available CPUs in two ways:

Static distribution The interrupt is delivered to one specific CPU, to a subset of CPUs, or to all CPUs at once (broadcast mode).

Dynamic distribution The IRQ signal is delivered to the local APIC of the processor that is executing the process with the lowest priority. However interrupts are distributed in a round-robin fashion among CPUs with the same task priority, adopting a technique called *arbitration*.

Linux makes use of three kinds of interprocessor interrupts:

`CALL_FUNCTION_VECTOR` It is used to force all CPUs to run a function passed by the sender. The corresponding interrupt handler is named `call_function_interrupt()`.

`INVALIDATE_TLB_VECTOR` It is used to force all CPUs to invalidate their TLB. The corresponding interrupt handler is named `invalidate_interrupt()`.

`RESCHEDULE_VECTOR` When a CPU receives this type of interrupt, the corresponding handler, named `reschedule_interrupt()`, limits itself to acknowledging the interrupt. Rescheduling is done automatically when returning from the interrupt.

Thanks to the following group of functions, issuing interprocessor interrupts (IPIs) becomes an easy task:

`send_IPI_all()` Sends an IPI to all CPUs (including the sender).

`send_IPI_allbutself()` Sends an IPI to all CPUs except the sender.

`send_IPI_self()` Sends an IPI to the sender CPU.

`send_IPI_mask()` Sends an IPI to a group of CPUs specified by a bit mask.

5.1 IST

In long mode, each IDT's entry have a very important field (3 bit) called **IST** (*Interrupt Stack Table*) which is completely absent on i386. **The IST is used to automatically switch to a new stack for events such trap or interrupts.** This mechanism unconditionally switches stacks when it is enabled and it can be enabled on an individual interrupt-vector basis using IST field in the IDT entry. This means that some interrupt vectors can use the legacy mechanism for stack-switch and others can use the IST mechanism.

There can be up to 7 IST entries per CPU and they are located on the Task State Segment (TSS). The IST entries in the TSS point to dedicated stacks; each stack can be a different size. **When an interrupt occurs and the hardware loads such a descriptor, the hardware automatically sets the new stack pointer based on the IST value, then invokes the interrupt handler**.

There are many stack provided by IST mechanism, one of them is the `DOUBLEFAULT_STACK`, which is used for the **Double Fault Exception** (`#DF`), invoked when handling one exception causes another exception. Using a separate stack allows the kernel to recover from it well enough in many cases.

5.2 Exception handling

Exception handlers have a standard structure consisting of three steps:

1. Save the contents of most registers (that is a CPU snapshot) in the Kernel Mode stack (this part is coded in assembly language);
2. Handle the exception by means of a high-level C function.
3. Exit from the handler by means of the `ret_from_exception()` function.

When a CPU receives an interrupt, it starts executing the code at the address found in the corresponding entry of the IDT. Registers is the first task of the interrupt handler and a pointer to the `pt_regs struct` (`include/asm-i386/ptrace.h`) is used to save register.

After the interrupt or exception is processed, `iret` assembly instruction is used to return program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception. In Protected Mode, the action of the `iret` instruction depends on the settings of the `NT` (nested task) flag: If the NT flag (EFLAGS register) is cleared, the `iret` instruction performs a far return from the interrupt procedure, without a task switch. Otherwise `iret` instruction performs a task switch (return) from a nested task

6 Slide '*virtual-file-system*'

The **Virtual File-system** (also known as **Virtual File-system Switch** or **VFS**) is a kernel **software layer that handles all system calls related to a standard Unix file-system**.

Linux VFS is capable to manage several kind of file-systems. Any file-system object, like file or directories, **is represented in RAM via specific data structures which are managed by any overlying kernel layer in a File System independent manner** (*FS independent part*). However, **each specific file-system implementation must translate its physical organization into the VFS's common file model**: it is possible **using a pointer for each operation**; the pointer is made to point to the proper function for the particular file-system being accessed (*FS dependent part*).

You can think of the common file model as object-oriented, where an object is a software construct that defines both a data structure and the methods that operate on it (although Linux is not written using any object-oriented language.)

6.1 Data Structure

To be able to manage a file system type we need several suitable data structure, which includes both the **object attributes** and a pointer to a table of **object methods**.

VFS's data structure are:

Superblock Object	Stores information concerning a mounted file-system. For disk-based file-systems, **this object usually corresponds to a file-system control block stored on disk**.
Inode Object	Stores general all functions and informations about a specific file. For disk-based file-systems, this object usually corresponds to a *file control block* stored on disk. Each inode object is associated with an inode number, which uniquely identifies the file within the file-system. *A filename is a casually assigned label that can be changed, but the inode is unique to the file and remains the same as long as the file exists.*
File Object	Stores information about the interaction between an open file and a process. This information exists only in kernel memory during the period when a process has the file open.
Dentry Object	Stores information about the linking of a directory entry (that is, a particular name of the file) with the corresponding file. VFS considers each directory a file that contains a list of files and other directories. Once a directory entry is read into memory, it is transformed by the VFS into a dentry object based on the dentry structure. **The kernel creates a dentry object for every component of a pathname that a process looks up**.

6.1.1 Superblock

All superblock objects **are stored in a circular doubly linked list** where the first element of this list is represented by the `super_blocks` variable. A super-block is represented by a `super_block` struct whose fields are described in following table:

Type	Field	Description
`struct file_system_type*`	`s_type`	File-system type
`struct super_operations*`	`s_op`	The methods associated with a superblock are called *superblock operations* and they are described by the `super_operations struct`. Each specific file-system can define its own superblock operation and the `super_operations` table is used to contain the address of them. They are invoked by VFS when it execute higher-level operations like deleting files or mounting disks like `read_inode`, `write_inode` etc. Generally all superblock operation implementations are provided by all possible filesystem types; however, the fields corresponding to unimplemented methods are set to NULL.
`struct dentry*`	`s_root`	Dentry object of the file-system's root directory
`struct list_head`	`s_dirty`	List of modified inodes

6.1.2 dentry

Usually dentry objects have no corresponding image on disk, and hence no field is included in the dentry structure to specify that the object has been modified. Dentry objects are stored in a slab allocator cache whose descriptor is `dentry_cache`;

Type	Field	Description
`struct inode *`	`d_inode`	Inode associated with filename.
`struct dentry *`	`d_parent`	Dentry object of parent directory.
`struct qstr`	`d_name`	Filename.
`struct list_head`	`d_child`	For directories, pointers for the list of directory dentries in the same parent directory.
`struct dentry_operations*`	`d_op`	The methods associated with a dentry object are called *dentry operations*. Although some file-systems define their own dentry methods, the fields are usually NULL and the VFS replaces them with default functions (like `d_delete(dentry)`, `d_compare(dir, name1, name2)`).

6.1.3 Inode

An inode object is represented by the `inode struct`:

Type	Field	Description
`struct list_head`	`i_dentry`	The head of the list of dentry objects referencing this inode
`struct inode_operations*`	`i_op`	The methods associated with an inode object are also called *inode operations* and they are stored by an `inode_operations` structure, which is populated by all possible inodes and filesystem types. Available functions are `create`, `mkdir`, `symlink` etc.
`struct file_operations*`	`i_fop`	Default file operations
`struct super_block*`	`i_sb`	Pointer to superblock object

6.2 File-systems

In Linux, *the code for a file-system actually may either be included in the kernel image or dynamically loaded as a module.* To handle all file-systems, VFS represents them as a `struct file_system_type` object, which is used to hold both **all file-system specific informations** and a **pointer to a function to be executed upon mounting the file system in order to read the super-block**.

Type	Field	Description
`const char *`	`name`	File-system name
`struct super_block * (*)( )`	`get_sb`	Method for reading a super-block. (Be aware that in newer kernel versions this field is called `mount`) **This is a file-system-type-dependent function that allocates a new super-block object and initializes it** (if necessary, by reading a disk when we manage a disk-based device). **Generally this function relies on a block-device driver of a device to instantiate the corresponding file system super-block in memory**. But be aware that if we use a RAM file systems, there is no need to read from a device, therefore this function is limited to in-memory instantiation of a fresh superblock.

All file-system-type objects are inserted into a singly linked list according to which the `file_systems` variable points to the first item. A spin lock called `file_systems_lock` protects the whole list against concurrent accesses.

During system initialization, the `register_filesystem()` function is invoked for every filesystem specified at compile time, inserting corresponding `file_system_type` object into the aforementioned list.

The `register_filesystem()` function is also invoked when a module implementing a file-system is loaded. In this case, the file-system may also be unregistered (by invoking the `unregister_filesystem()` function) when the module is unloaded.

6.2.1 Filesystem Mounting

The directory on which a file-system is mounted is called the **mount point**. (*remember that the root directory of a mounted file-system hides the content of the mount point directory of the parent file-system, as well as the whole sub-tree of the parent file-system below the mount point.*)

Is possible to mount the same file-system several times and, if it is mounted n times, its root directory can be accessed through n mount points, one per mount operation. **Although the same file-system can be accessed by using different mount points, there is only one super-block object for all of them, no matter of how many times it has been mounted.**

For each mount operation, the kernel must save in memory several information, like the mount point, the mount flags or the relationships between the file-system to be mounted and the other mounted file-systems. **Such informations are stored in a mounted file-system descriptor of type** `struct vfsmount`.

The `struct vfsmount` data structure, which from Linux kernel 4.8 is flagged with the `__randomize_layout` annotation, has several fields including:

Type	Field	Description
`struct vfsmount *`	`mnt_parent`	Points to the parent file-system on which this file-system is mounted.
`char *`	`mnt_devname`	Device file-name.
`struct list_head`	`mnt_mounts`	Head of a list including all file-system descriptors mounted on directories of this filesystem.
`struct list_head`	`mnt_child`	Pointers for the `mnt_mounts` list of mounted file-system descriptors.
`struct namespace *`	`mnt_namespace`	Pointer to the name-space of the process that mounted the file-system.
`struct super_block *`	`mnt_sb`	Points to the super-block object of this file-system.
`int`	`mnt_flags`	Flags that specify how some kinds of files in the mounted file-system are handled. For instance `MNT_NOEXEC` flag is used to disallow program execution in the mounted file-system.

6.2.2 `rootfs`

Linux makes use of a system's **root file-system** (`rootfs`) *which is directly mounted by the kernel during the booting phase and that holds the system initialization scripts and the most essential system programs.* The `file_system_type` struct keeps meta-data for `rootfs` is allocated and initialized statically thorough `init_rootfs()` function within `fs/ramfs/inode.c`.

6.2.3 The VFS startup and `rootfs` mounting

VFS is initialized by Linux through `vfs_caches_init()` function which is a crucial to mount `rootfs` and all other required file-systems. (*However, in principles, the Linux kernel could be configured such in a way to support no file-system. In this case, any task to be executed needs to be coded within the kernel*).

Mounting process is performed by the following functions (invoked by `vfs_caches_init()`):

`init_rootfs()` function registers the special file-system type *rootfs* by setting a `struct file_system_type` variable called `rootfs_fs_ty`

`init_mount_tree()` This function executes the following operations:

1. The `do_kern_mount()` function is invoked passing to it the string "rootfs" as filesystem type; **this function checks the file-system type flags to determine how the mount operation is to be done**. `do_kern_mount()` (**it is used to mount generic file-systems**) performs several operations like:
   - execute the file-system-dependent function to allocate a new super-block and to initialize it (invoking `get_sb` function).
   - Initializes the dentry object corresponding to the root directory of the filesystem, and increases the usage counter of the dentry object (`mnt_root` field is used)
   - Initializes the name-space field with the name-space of the current process (that is `mnt->mnt_namespace` field is set with the value in `current->namespace`.) (not done when initialize `rootfs`)

   Finally, `do_kern_mount()` function returns a the address of the mounted filesystem descriptor,
2. Allocates and initialize a `namespace` object for the namespace of process 0, and inserts into it the mounted file-system descriptor previously allocated. During system initialization, the name-space field of every other process in the system is set to the address of the namespace object of process 0 (**By default, all processes share the same, initial namespace**) (it is also initialized the `namespace->count` usage counter).
3. Sets the root directory and the current working directory of process 0 (the idle process) to the root filesystem using `set_fs_pwd()` and `set_fs_root()` function.

6.2.4 Namespaces

The list of mount points along directory tree is called name-space. *In Linux, every process might have its own tree of mounted file-systems, that is the so called name-space of the process.*

Usually most processes share the same name-space, which is the tree of mounted file-systems that is rooted at the system's root file-system and that is used by the `init` process.

However, a process gets a new name-space if it is created by the `clone()` system call with the `CLONE_NEWNS` flag set.

If a parent process creates a child process without the `CLONE_NEWNS` flag, its names-pace is inherited by its children.

When a process mounts or unmounts a file-system, it only modifies its names-pace. Therefore, the change is visible to all processes that share the same name-space, and only to them (except if the mount operation is tagged with SHARED).

6.2.5 Structure randomization

Fields in a C structure are laid out by the compiler in order of their declaration. One technique for attacking the kernel is to **overwrite specific fields of a structure with malicious values** and, when the order of fields in a structure is known, it is trivial to calculate the offsets where sensitive fields reside.

A useful type of field for such exploitation is the function pointer, which an attacker can overwrite with a location containing malicious code that the kernel can be tricked into executing.

The `randstruct` plug-in randomly rearranges fields at compile time given a randomization seed. When potential attackers do not know the layout of a structure, it becomes much harder for them to overwrite specific fields in those structures providing extra protection to the kernel.

To get structure randomization working in the kernel, the structures marked for randomization need to be tagged with the `__randomize_layout` annotation. Usually, structures that only contain function pointers are a big target for attackers and reordering them is unlikely to cause problems elsewhere; therefore **data structures consisting entirely of function pointers are automatically randomized**. This behaviour can be overridden with the `__no_randomize_layout` annotation.

Naturally, **compiler support is necessary to get this feature to work**. Since Linux kernel 4.8, this plug-in is been used.

6.3 Relationship with processes

To represent the interactions between a process and a file-system, Linux use the `fs_struct` data structure (defined in `include/fs_struct` and **each process descriptor has an `fs` field** that points to the process `fs_struct` structure.

Some `fs_struct` fields are described below:

Type	Field	Description
`atomic_t`	`count`	Number of processes sharing this table
`struct dentry *`	`root`	Dentry of the root directory
`struct dentry *`	`pwd`	Dentry of the current working directory
`struct vfsmount *`	`rootmnt`	Mounted filesystem object of the root directory
`struct vfsmount *`	`pwdmnt`	Mounted filesystem object of the current working directory

In Linux 3.xx/4.7, the `fs_struct` data structure is slightly different: the `root` and `pwd` fields are now of `struct path` type. From kernel 4.8, this data structure is flagged as `__randomize_layout`.

The `files_struct` data structure represents the **file descriptor table**, whose address is contained in the `files` field of the process descriptor, used to specify which files are currently opened by the process (*is also said that this table builds a relation between an I/O channel (a numerical ID code, used as a key) and an I/O object*). Some fields are:

Type	Field	Description
`atomic_t`	`count`	Number of processes sharing this table
`struct file **`	`fd`	Pointer to array of `struct file` object pointers. The size of the array is stored in the `max_fds` field. For every file with an entry in the `fd` array, the array index is the *file descriptor*. Usually, the first element (index 0) of the array is associated with the **standard input** of the process, the second with the **standard output**, and the third with the **standard error**. A process cannot use more than `NR_OPEN` (usually 1048576) file descriptors.
`fd_set *`	`open_fds`	Pointer to open file descriptors.

We said that the file descriptor table contains a pointer to an array of `struct file` objects. A `file` object (*the session data of the file descriptor table*) describes how a process interacts with a file it has opened and it is created when the file is opened. *Notice that file objects have no corresponding image on disk.* (in newer version of kernel, `file` data structure is flagged with `__randomize_layout` annotation) `file` fields are:

Type	Field	Description
`struct dentry *`	`f_dentry`	dentry object associated with the file.
`struct file_operations *`	`f_op`	Pointer to file operation table.
`atomic_t`	`f_count`	File object's reference counter which counts the number of processes that are using the file object.
`mode_t`	`f_mode`	Process access mode.
`loff_t`	`f_pos`	Current file offset (file pointer).

6.4 Device Files

Unix-like operating systems are based on the notion of a file. This is true also for I/O devices which are treated as special files called **device files**; **thus, the same system calls used to interact with regular files on disk can be used to directly interact with I/O devices**.

Device files can be of two types:

Block Device In this devices type, the data can be addressed randomly, and the time needed to transfer a data block is small and roughly the same.

Char Device The data of a character device either cannot be addressed randomly, or they can be addressed randomly, but the time required to access a random datum largely depends on its position inside the device.

A device file is usually a real file which inode **includes an identifier of the hardware device corresponding to the character or block device file**. This identifier consists of a pair of numbers:

Major number It is used to identify the device type. Traditionally, all device files that have the same major number and the same type share the same set of file operations, because they are handled by the same device driver (*the operations for managing the device are retrieved via the device driver tables*)

Minor number It is used to identify a specific device among a group of devices that share the same major number. For instance, a group of disks managed by the same disk controller have the same major number and different minor numbers.

In x86 machines, device numbers are represented as bit masks stored into `kdev_t i_rdev` field of `struct inode`. In particular, major number corresponds to the least significant byte within the mask, while the minor correspond to second least significant byte. The macro `MKDEV(ma,mi)` (defined in `include/linux/kdev_t.h`), can be used to setup a correct bit mask by starting from the two numbers.

The `mknod()` system call is used to create device files. It receives the name of the device file, its type (`S_IFREG`, `S_IFCHR`, `S_IFBLK`, `S_IFIFO`), and the major and minor numbers as its parameters. Device files are usually included in the `/dev` directory.

Usually, a device file is associated with a hardware device (such as a hard disk) or with some physical or logical portion of a hardware device.

6.4.1 Device Drivers

A device driver is the set of kernel routines that makes a hardware device respond to the programming interface defined by the canonical set of VFS functions that control a device (that is `open`, `write`, `read` and so on...)

The actual implementation of all these functions is delegated to the device driver. Because each device has a different I/O controller, and thus different commands and different state information, most I/O devices have their own drivers.

Each system call issued on a device file is translated by the kernel into an invocation of a suitable function of a corresponding device driver. **To achieve this, a device driver must register itself**. In other words, registering a device driver means allocating a new `device_driver` descriptor.

A **character device driver** is described by a `cdev` structure, whose main fields are:

Type	Field	Description
`dev_t`	`dev`	Initial major and minor numbers assigned to the device driver.
`struct file_operations *`	`ops`	Pointer to the file operations table of the device driver.
`struct list_head`	`list`	Head of the list of inodes relative to device files for this character device

To keep track of which character device numbers are currently assigned, the kernel uses a hash table `chrdevs`, which contains intervals of device numbers. **Two intervals may share the same major number, but they cannot overlap, thus their minor numbers should be all different**.

The table includes up to 255 entries every of which points to the first element of a collision list ordered by increasing major and minor numbers.

For assigning a range of device numbers to a character device driver, is possible to use both the `register_chrdev_region()` function, which assigns an arbitrary range of device numbers (*it receives as its parameters the requested major number major, the name of the device driver name, and a pointer to a table of file operations specific to the character device files in the interval*), and the `register_chrdev()` function which assigns a fixed interval of device numbers including a single major number and minor numbers from 0 to 255 (*it receives as its parameters the requested major number major, the name of the device driver name, and a pointer to a table of file operations specific to the character device files in the interval.*)

6.5 `procfs`

The `proc` file-system is a **pseudo-file-system** provided by the Linux kernel which is **designed to allow User Mode applications to access kernel internal data structures**, exporting information about various kernel subsystems, hardware devices, and associated device drivers **via common I/O operations on virtual files**. It is automatically mounted at `/proc`.

`/proc` includes non-process-related system informations like:

`cpuinfo` contains information about the processor at boot time.

`meminfo` contains information about the memory usage.

`version` contains the kernel version information.

`kcore` contains the entire RAM contents as seen by the kernel

6.6 `sysfs`

sysfs is a **pseudo file-system** similar to `proc`. `sysfs` is compiled into the kernel by default depending on the configuration option `CONFIG_SYSFS` (visible only if `CONFIG_EMBEDDED` is set).

A goal of the sysfs filesystem is to expose the hierarchical relationships among the components of the device driver model.

- Relationships between components of the device driver models are expressed in the sysfs filesystem as symbolic links between directories and files.
- The main role of regular files in the sysfs filesystem is to represent attributes of driv- ers and devices.
- The related top-level directories of this filesystem are used to represent several kernel object like block, device, bus and so on.