Introduction

An **operating system** is the low level program that sits **between the hardware and user applications**. Its functions include:

1. Manage computer system's **resources:**
    a. Processors
    b. Memory
    c. I/O devices
    d. Data and storage
2. Provide **user interface**
3. Provide **services for applications**.

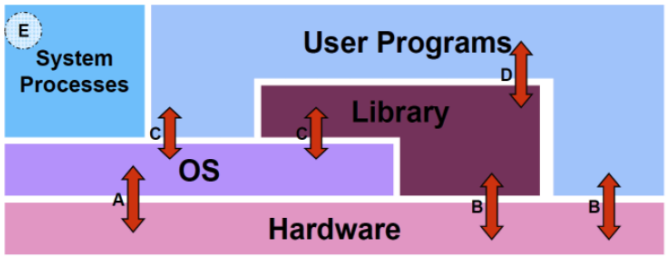
Other key services of OS:

- Hardware abstraction
    - Regulate access to hardware
    - Drivers
    - Book-keeping services
- Protection
    - User and root access modes
- Sharing and data exchange
    - IPC
    - Networking
- Caching
    - TLB (virtual memory)
    - Processor cache
    - Disk cache
- Interrupt handling
    - Interrupts (asynchronous)
    - Traps (synchronous)

Types of OS

Batch, interactive, realtime…

Structures of OS



- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes
  - Provide high level services, usually part of OS

Note that B (between user program and hardware) is mainly for **machine instructions**! You **cannot access** hardware resources directly without the OS as it is very dangerous.

Monolithic OS

Properties:

- **Single process**
- Single address space (kernel space)
- Single binary file loaded at boot time (excluding drivers?)

An example would be **Linux**.

Microkernel

Properties:

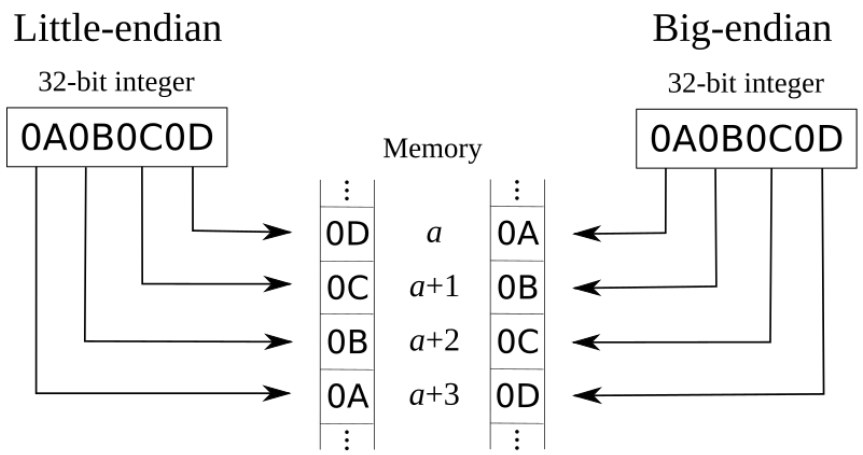
- Small kernel supporting minimal services:
  - IPC, virtual memory, process scheduling, interrupt handling, protection
  - Better security (principle of least privilege)
- Services not included in kernel:
  - Drivers, process management (bookkeeping), memory management (bitmaps), file systems are all stored in **user mode**.
- Services communicate using IPC:
  - Causes **overhead** in performance.

Exokernel (experimental)

Give applications direct access to hardware via libraries.

Intel Architecture

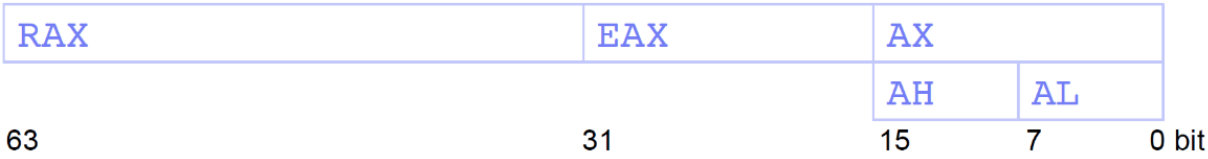
Perspective: **Ascending** address
Little endian: LSB first
Big endian: MSB first



Registers and sub registers

Registers contain subregisters. For example:

- RAX contains EAX, which contains AX.
- RBX contains EBX, which contains BX.



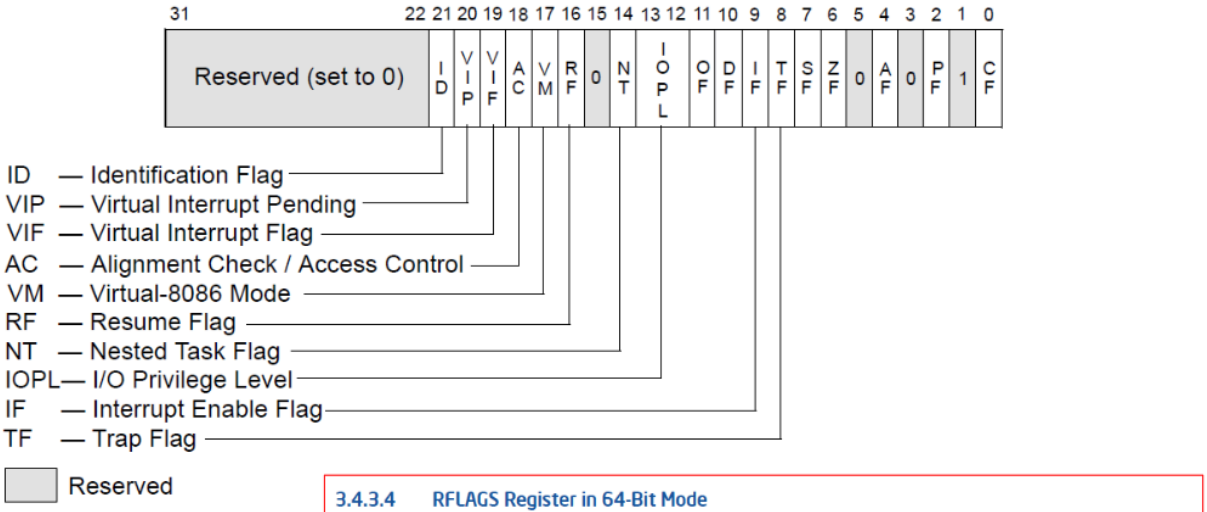
32-bit: EAX (Accumulator), EBX (Base Index), ECX (Counter), EDX (Data), ESI (Source Index), EDI (Destination Index), ESP (Stack Pointer), EBP (Base Pointer), EIP (Instruction Pointer)

16 bit: CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES, FS, GS

64-bit: RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, RIP, R8, ..., R15

EFLAGS

EFLAGS is a register that stores the **current state** of the machine.

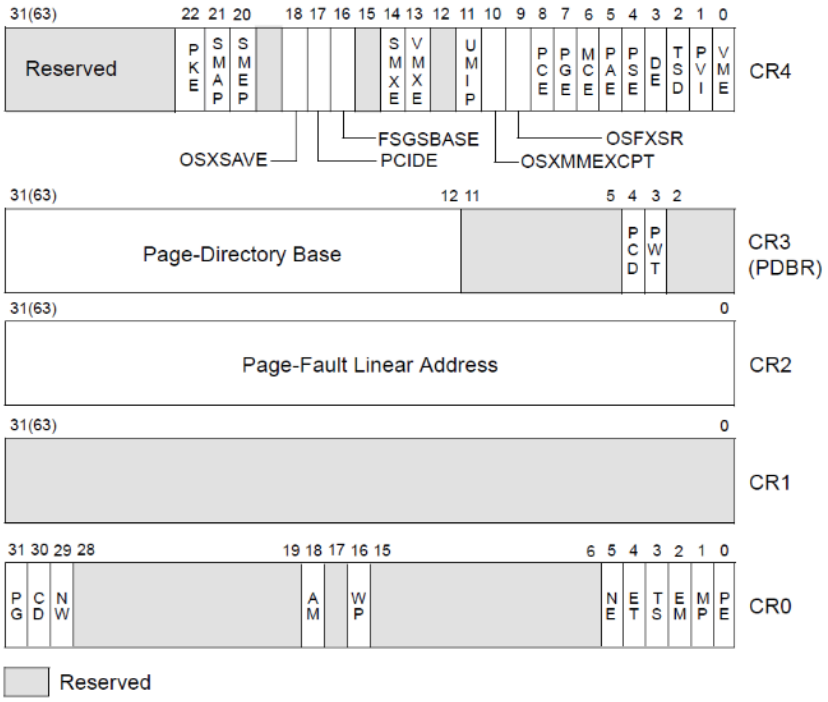


3.4.3.4 RFLAGS Register in 64-Bit Mode

In 64-bit mode, EFLAGS is extended to 64 bits and called RFLAGS. The upper 32 bits of RFLAGS register is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS.

Control Registers

Changes or controls the **general behavior** of the CPU, such as page table address, segment table address, etc.



Operating Modes

- Real-address mode: usually used during the boot process **before virtual memory is set up**.
- Protected mode: normal operation with virtual memory and privilege rings.
- System Management mode: Suspends normal execution and executes another software system (eg. firmware) in the **system management RAM** (separate address space).
- Virtual-8086 mode: Executes real-mode executions while the processor is running in protected mode.
- Intel 64 IA-32e mode: Executes both 32 and 64-bit instructions.

Memory Address

Flow (segmentation + paging):

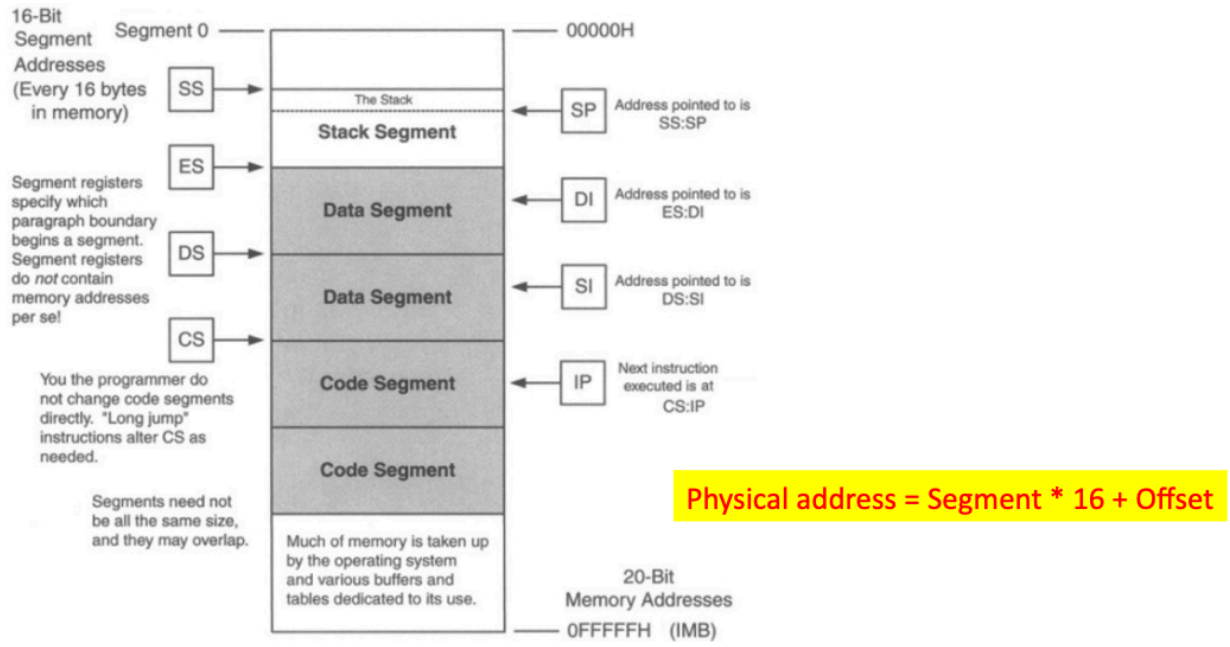
1. (Segmentation) Use **logical address** and global descriptor table to find the **linear address**.
2. (Paging) Use **linear address** and multilevel page tables to find the **physical address**.

Flat model:

- Directly uses linear address.

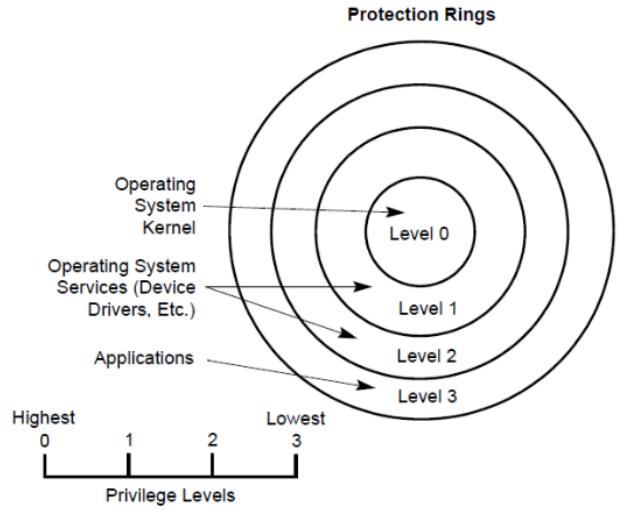
Real-Address mode model:

- Uses equal-sized “segmentation” without paging.
- Real address = Segment register * 16 + Offset



Protection rings

Order of protection: Applications > Drivers / OS Services > Kernel



Current privilege level is stored in the 2 LSB bits in the **CS register** (which also stores the address of the code segment).

You can only **switch from Ring 0 to 3** using **syscalls or interrupts (for hardware)**.

Instruction Encoding

ModRM Byte

1. **MOD** field specifies **addressing mode,** eg. using registers, displacement (using a constant), SIB.
2. **REG** field specifies source or destination **register**.
3. **R/M** field may contain a **register or an addressing mode** (such as a displacement).

SIB Byte

Uses scale, index and base for **complex addressing**.

Examples:

- add %ebx, %eax -> Only uses the ModRM byte.
- mov 0x20 (%esp), $esi
    - Uses complex addressing, hence requires SIB byte. As displacement is only one byte (0x20), this means MOD = 01 and R/M = 100 (reference to table).

For **64 bit encoding**, extra bits allow more registers to be used:

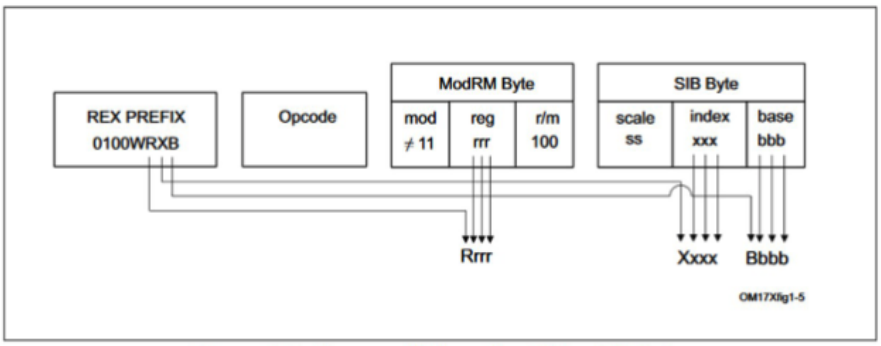


Figure 2-6. Memory Addressing With a SIB Byte

x86 Assembly Syntax:

1. Intel: Destination before source.
2. AT&T: Source before destination.

Note: It is **impossible to do memory-memory** transfers with a single instruction.

Address computation

Expression	Computation	Address
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%edx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Note that the offset can only be 1, 2, 4 or 8.

Condition codes

Some bits are used to manage control flow, implicitly set by **arithmetic operations**:

1. Carry Flag - Tracks unsigned overflow
2. Zero Flag - Set if output == 0
3. Sign Flag - Set if output < 0
4. Overflow Flag - Set if two's complement overflow (signed overflow for 2's complement)

Condition codes can also be explicitly set, such as using the **cmpl b, a** instruction (which is effectively computing a - b).

IA32 Stack

Grows “**downwards**” towards the **lower addresses**. %esp represents the address of the **top element**.

Functions work using (push/pop + double jump):

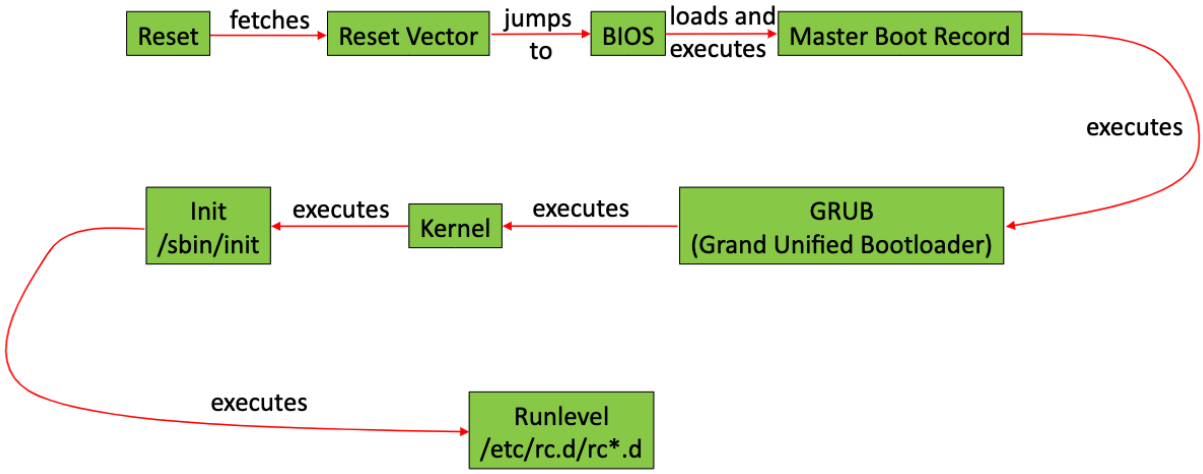
1. **call label** (pushes return address on stack and jumps to **label**)
2. **ret** (pops address from stack and jumps to **return address**)

Caller and Callee Saved Registers

Caller saved: Caller needs to save the registers **before calling**.
Callee saved: Callee needs to save and **restore** the registers **before returning**.

Boot Process

Summary of flow:



Firmware

A **low level software** that works directly with hardware. It is often stored in **non-volatile memory**:

- Does not require power
- Has slow reads, slower writes.
- Writes cannot happen for data during execution.
- Updating is slow.

Start of boot process (only hardware checks)

1. Hardware ensures power sources are stable and initializes (known as **power sequencing**).
2. (For multiprocessor systems) Hardware protocol is used to determine the bootstrap processor, which will execute the boot sequence.
3. The processor core's PC will be set to the **reset vector**. The reset vector will be the very first instruction fetched.
4. The first instruction jumps into the real entry of the **BIOS code**.

BIOS

Known as Basic Input/Output System. It is a **firmware** for hardware initialization and booting that resides in non-volatile memory. It is succeeded by **UEFI**.

Flow of BIOS code (occurs in **real mode**, memory limited to 1 MB):

- Verify registers, basic components (such as DMA, timer, interrupt controller), verify memory, verify integrity of BIOS code, initialize BIOS, select devices available for booting.
- Essentially, contains more hardware/software checks and initializations.

The **upper memory area** (between 640 KB - 1024 KB) stores the video adapter memory, device options ROMs and special RAM shared with physical devices.

Segment Descriptor Cache

Normal Use Case

Each time a **segment register** is loaded, the **segment descriptor information** is loaded (eg. base, limit, attributes) into hidden registers as a **cache**. Without this, every memory access will result in more memory access.

Reason: No change in the **segment register** => No change in GDT values.

BIOS Use Case

Allows access to more (and possibly the entire) memory by overriding the segment limit.

"Unreal" Mode

Meant to help utilize larger than 20 bit addresses.

Flow:

1. Switch to protected mode.
2. Set up **descriptors, selectors and GDT**.
3. Set **segment limit to 4GB**.
4. Set privilege level appropriately.
5. Revert back to real mode.

The **segment limit is stored in the cache**, allowing the system to access more than 20 bits.

BIOS Software Setup

Note: Everything is initially **read-only in the BIOS**.

Load a set of minimal copy of:

1. Interrupt descriptor table
2. Global descriptor table
3. Task state segment
4. (Optional) Local descriptor table - Similar to GDT but **local to task/thread**.

Initialize:

1. Critical control registers
2. **Memory type range registers**
    a. Set to “no evict mode” to enable **cache to act as memory**.
    b. Remember to undo this when memory is loaded.
3. Memory itself

After memory is available, **BIOS can load into the OS**:

1. Initialize secondary storage devices.
2. Find the **master boot record** and verify its integrity.
3. Load the **bootstrap loader** (code section of MBR).

Disk partitions

- Can be used for different OS, swap disk, user files, etc.
- Protects different areas of the disk.
- Reduce seek time as files are more localized.

GRUB

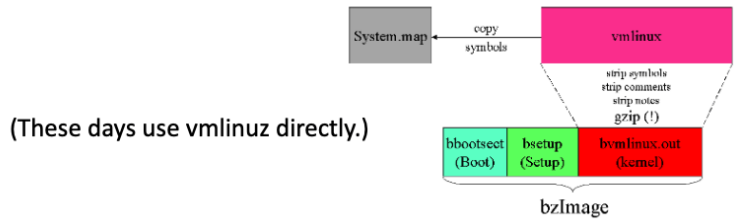
Loading process: boot.img -> diskboot.img -> rest of GRUB -> grub_main()

grub_main() initializes the console, gets the base address for modules and sets the root device. Fills in the **Linux kernel header** which specifies the Linux version being used.

Linux kernel (a single file loaded by GRUB)

- vmlinux - a statically linked executable file that contains the entire kernel
- vmlinuz – compressed vmlinux
- zImage – old kernel below 512KB that is loaded into low memory
- bzImage – larger kernel loaded at above 1MB

(These days use vmlinuz directly.)



Kernel boots (still in real mode)

- Set up memory
  - Set up stack and bss
  - Jump to main.c
  - Switch to protected mode
  - Build page tables
- Decompress kernel
- Mounts **initrd** (minimal filesystem that fits in RAM)
  - This is required for the rest of the OS boot process to make sense of files in the disk. When it is done, **pivot_root** is done to switch to the real file system.
- Run **init** (the first user level process)
  - Load all necessary drivers
  - Switch back to disk file system
- Alternatively, there is **systemd** which starts processes in parallel and has a simpler API.
  - Everything is viewed as a **DAG of units**
  - Sets up multiple daemons (journald, networkd, logind, udevd)

UEFI

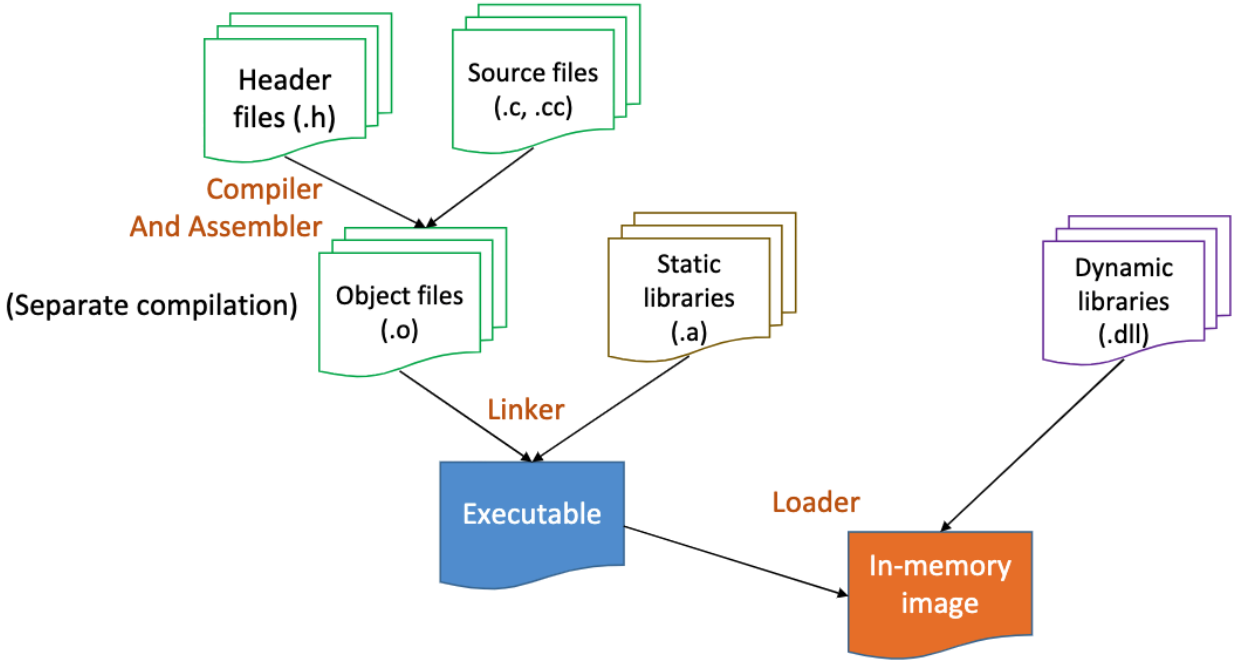
The successor to BIOS, with more capabilities such as:

- Booting from large disks
- GUID Partition Tables

- Better GUI
- No byte limit on bootstrap loader of MBR
- New partition structure
- Authenticates loaded software with keys burnt into ROM
- And more…

Linking and Loading

High-level overview:



Summary:

- Compile time:
  - Compiler + Assembler -> **object files**
  - Linker - Links all the **object files and static libraries** into an executable.
  - This helps with better **organization** and sharing of reused code.
- Run time:
  - Loader - Runs executable with **dynamic libraries**, and constructs the memory image.

ELF (Executable and Linking File Format)

Files types:

1. Relocatable (object files):
   a. Must be processed by linker before running/
   b. Have **section header tables** (points to sections).
2. Executable:

    a. Contain information to be **conveyed to the OS**:
        i. Header Info
        ii. Code & Data
        iii. Symbol table
    b. Everything done except for **shared library symbols**.
    c. Have **program header tables** (points to segments).
3. Shared object:
    a. Shared libraries (as it can be shared by multiple programs at load time)
    b. Have **section and program header tables**.
4. Core file - For core dumps: recorded state of program that crashed (eg. memory, registers, etc)

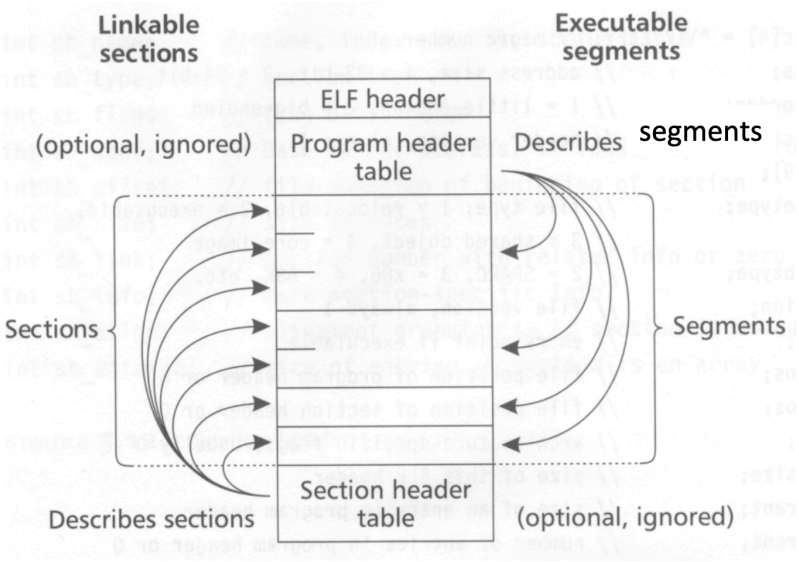
A single segment usually consists of **several sections**.

Before linking

ELF is treated as logical **sections** (intended for further processing).

After linking

ELF is treated as a set of **segments** (intended to be **mapped to memory**). Segments contain sections.



ELF Header (offset zero)

Defines the offset of the **section** and **program header tables**:

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

4-byte magic number is “0x7F” followed by the string “ELF”.

Relocatable Files (or object files)

Is a **collection of sections**, where each section contains a **single type of information**, such as program code, read-only data, relocation entries, symbols, etc.

Every **symbol’s address** is defined **relative to a section**; The section is like the “address space” for the symbol.

Section Header

Contains information to describe a section:

```
typedef struct
{
  Elf32_Word    sh_name;                /* Section name (string tbl index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds table */
} Elf32_Shdr;
```

Key fields to define a section type

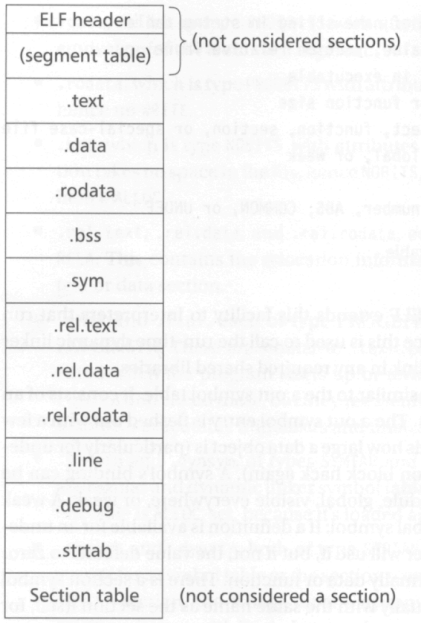
sh_type

- like PROGBITS, NOBITRS, SMTAB, DYNSYM, etc.

sh_flags

- specifies permissions like WRITE, ALLOC, EXECINSTR.

These 2 properties **uniquely identify** each section such as text, data, bss, etc.



Names and Symbols

Multiple tables are used in the compilation and execution process to manage names and symbols.

String Table

An **index** is used to **reference a string**. These strings are used in object files to represent symbol and section names. It is **separate from the symbol table**.

Symbol Table

A **symbol table index** is used to access information needed to locate and relocate a program's symbolic definitions and references.

```
typedef struct
{
  Elf32_Word    st_name;                /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;               /* Symbol value */
  Elf32_Word    st_size;                /* Symbol size */
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf32_Section st_shndx;               /* Section index */
} Elf32_Sym;
```

The section relative in which the symbol is defined. (e.g., the function entry points are defined relative to .text)

Dynamic Symbol Table

.dynsym is a subset of **.sym tab**. In other words, the dynamic symbol table is a subset of the symbol table.

Differences:

- Symbol table is needed at **static link time**. However, not all of the information is needed at runtime, hence it's not allocated in process memory.
- Dynamic symbol table is needed at **runtime**. This is because the information is needed to execute the program.

Symbol Lookup

1. Use the index of the symbol table (if you have it).
2. Use the **.gnu.hash** sections:
    a. Header (bloom filter and hash table parameters)
    b. Bloom Filter
    c. Hash Buckets
    d. Hash Values

Bloom Filter

- Guarantees **true negatives**.
- Elements can be added but cannot be removed.
- More elements -> Higher probability of false positives.

Flow

1. Find index into the Bloom array using the hash.

2. Create bitmask (essentially two 1's positioned based on the hash).
3. Test / Add the bitmask.

If found, you need to use the hash **buckets**.

Hash Buckets and Values

It is essentially a **hashmap** for the dynamic symbol table, where the chain is represented with an array.

There are **3 data structures** being used together:

1. Dynamic symbol table
    a. Contains the actual symbol.
    b. **The** (contiguous) **sequence of symbols is the same as the sequence in the chains in the GNU Hash Table**.
2. Hash Buckets
    a. Contains location of **first symbol in each bucket** inside the **dynamic symbol table**.
    b. Allows you to use the **index of the chain** to find the **corresponding symbol in the .dynsym**. This works as their sequences are the same.
3. Hash Values
    a. Also known as the **GNU Hash Table** or **DT_GNU_HASH**
    b. It maps the n-th bucket to its chain.
    c. Each chain is simply a **contiguous sequence of hashes** in an array.
    d. The last bit of each hash (stopper bit) in the chain is used to **determine the "end" of the chain**.

Flow

1. Find value of **N**: The N-th entry of the hash bucket.
2. Scan through the chain of the N-th bucket to **find the hash**. Comparing hashes is faster than comparing strings.
3. If found:
    a. Compare with the corresponding symbol in the .dynsym.
    b. If equal => Found!

```
nbuckets = 4      (because I decided that there will be four buckets)
symoffset = 1    (STN_UNDEF is not a part of the hash table)
bloom_size = 2   (because I decided that 16 byte bloom filter is sufficient)
bloom_shift = 5  (again, just because I can)

ix  bucket[ix]  name of first symbol in chain
--  ----------  ----------------------------
 0  1           cfsetispeed
 1  5           uselib
 2  8           freelocal
 3  13          getspen

Note that:
- symbol table is sorted by bucket
- chain[ix] is the same as hash but with set/cleared lowest bit

       SYMBOL TABLE              |              GNU HASH TABLE
                                 |
    name =                       |    hash %             bloom  bloom bits
ix  symtab[ix].st_name    hash   | ix nbuckets chain[ix]  word   #0    #1
--  ------------------  -------- | -- -------  ---------- -----  ---   ---
 0  <STN_UNDEF>                  |
 1  cfsetispeed         830acc54 |  0    0      830acc54    1    20    34
 2  strsigna            90f1e4b0 |  1    0      90f1e4b0    0    48    37
 3  hcreate_            4c7e3240 |  2    0      4c7e3240    1     0    18
 4  endrpcen            b6c44714 |  3    0      b6c44715    0    20    56
 5  uselib              2124d3e9 |  4    1      2124d3e8    1    41    31
 6  getttyen            fff51839 |  5    1      fff51838    0    57     1
 7  umoun               1081e019 |  6    1      1081e019    0    25     0
 8  freelocal           e3364372 |  7    2      e3364372    1    50    27
 9  listxatt            ced3d862 |  8    2      ced3d862    1    34     3
10  isnan               0fabfd7e |  9    2      0fabfd7e    1    62    43
11  isinf               0fabe9de | 10    2      0fabe9de    1    30    14
12  setrlimi            12e23bae | 11    2      12e23baf    0    46    29
13  getspen             f07b2a7b | 12    3      f07b2a7a    1    59    19
14  pthread_mutex_lock  4f152227 | 13    3      4f152226    0    39    17
15  getopt_long_onl     57b1584f | 14    3      57b1584f    1    15     2
```

Advantages of GNU Hash Table

It simply provides much fast loading time:

1. **Multiple levels of filtering** such as **bloom filter** and **comparison of hash**, which helps to filter true negatives early and save time.
2. Contiguous chains likely make use of **caching**, which is better than using some linked list.

Name Mangling

It is noteworthy that names in the symbol tables are usually different from the source code. This is done to avoid **name collisions, name overloading and type checking**.

Relocation

Relocation Table

The process of connecting symbolic references with symbolic definitions. Essentially, it tells the linker **where and what** to replace in the file. It contains **relocation entries**.

Relocation Entry

Field	Purpose
r_offset	The location to replace the symbol: 1. For relocatable files, it is a byte offset from the beginning of the section. 2. For executables and shared objects, it is a virtual address.
r_info	Contains two values: 1. **Symbol table index** 2. **Relocation type** - defines the way to compute the address. For example, in the instruction “mov disp(%rip), %eas”, where **disp is the value to be replaced**: a. It is an offset from %rip, **which is 4 bytes ahead**. Hence, you have to subtract 4 from the address of the symbol. b. It is also an **offset**. Hence, you need to find the difference between the symbol (after subtracting 4) and the address where it is replaced. Essentially, it can depend on the instruction and which value you’re replacing, which affects what value you’re going to replace the offset with. Hence, there will be different ways to compute an address.
r_addend	It is a constant value to be added to compute the value stored in the relocation field. It may be used based on the **relocation type**. For the example above, the addend = -4

For example, the first entry of the relocation table below means that the value at offset 0x6 should be replaced by the address of symbol **example_of_global_var**.

```
Relocation section '.rela.text' at offset 0x5d0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000006  000900000002 R_X86_64_PC32     0000000000000000 example_of_global_var - 4
00000000000d  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
000000000017  000b00000002 R_X86_64_PC32     0000000000000000 printf - 4
```

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # a <main+0xa>
   a:   89 c6                   mov    %eax,%esi
   c:   bf 00 00 00 00          mov    $0x0,%edi
  11:   b8 00 00 00 00          mov    $0x0,%eax
  16:   e8 00 00 00 00          callq  1b <main+0x1b>
  1b:   5d                      pop    %rbp
  1c:   c3                      retq
```

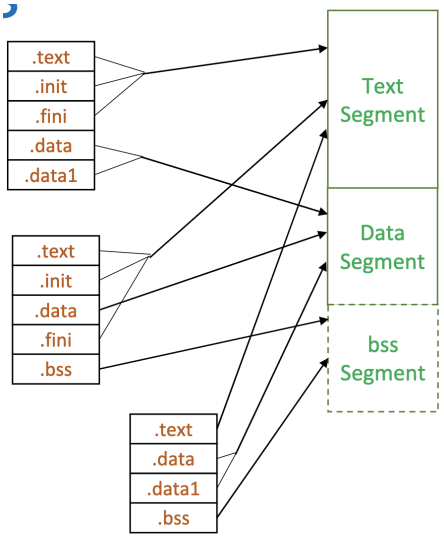
Allocation in Linking

Each object file:

- Is compiled independently and differently.
- Assumed to start at address 0.

They need to ultimately be combined into a single executable by:

1. collecting sections of the same name into a single section
2. place these combined sections into segments



Executable Files

Usually only has a few segments (for the final process image).

Program Header

```
typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;
```

Type	Purpose
PT_LOAD	Loadable segment (eg. the normal stuff)
PT_DYNAMIC	Dynamic linking information. It contains information needed by the resolver such as GOT, symbol table, etc.
PT_INTERP	Contains location and size of a null-terminated path name. It is used for the program interpreter in dynamic linking.

Link Scripts

Allows you to manually start sections at specific virtual addresses.

Dynamic Linking

It is motivated by the usage of **shared libraries**. It is important to know that **shared libraries can also use shared libraries**. Hence, even the shared libraries themselves use dynamic linking!

Shared Libraries (.so)

A **single copy** of commonly used routines (in memory) for all code in a system:

- Easy to maintain
- Save space
- Resolved at **load time**: Process copies in the shared library into their address space.

Advantages:

1. Faster load time: As shared library code may already be in memory (don't need to read from disk).
2. Better run-time performance: Less likely to page out frequently used code.
3. Easier for shared libraries to be updated.

Disadvantages:

1. Need to "glue code"
2. Reduced locality of references
3. May impact paging
4. Changes in libraries may break some applications (which means you need to recompile).

Static Libraries (.a)

Simply a concatenation of object files (relocatables).

Global Offset Table

All external addresses used by the application. It is fixed up during load time.

Special Entries	Purpose
GOT[0]	Address of **.dynamic segment**.
GOT[1]	Pointer to a linked list corresponding to the **dynamic segment of each shared library** linked with the program.
GOT[2]	Address of **symbol resolution function** (the loader itself).

Procedure Linkage Table

The **compiler** ensures that every call to the same dynamically linked routine will go to the **same PLT entry**.

Each (64-bit) entry is an **external function reference** which contains 3 instructions:

Component	Description
jmpq *GOT[entry for function]	Used as a "trampoline".
pushq <GOT entry number of function>	Starts from 0, i.e. GOT[3]
jmp PLT[0]	

.rela.plt section (found in **.dynamic**) of the executable, which contains the GOT, has the appropriate relocation information to **fix up the GOT at runtime**.

“BIND NOW”

Flow:

1. Find and load in all dynamic libraries.
2. Fix up the GOT entries of the main application.

RELRO

Known as “**read only relocation**”. Loader will change GOT protection to **read-only** to protect against loader memory area overwrites.

Position Independent Code

Motivation
Shared libraries may be mapped to **different absolute addresses for each program**, this may result in issues with **accessing global information** of the shared libraries as they are fixed references.

Why not set a system-wide address?
It is also difficult to fix system-wide where a shared object should reside. It is even harder due to **address space randomization**.

Solution
PIC ensures that the code can execute properly regardless of its absolute address by **making all the memory references relative**:

- The **loader** uses PLT and GOT to indirectly obtain **global information** (data or procedures). This makes the references **relative**.
- There is no issue for local data and branching information as they are already relative (referenced by offsets).

Dynamic Loading

Allows loading of new shared libraries at **runtime** using the **dynamic loading API**. The API is accessed via the program code.

Loader (in Linux)

Invokes **execve()** syscall, which eventually calls **load_elf_binary()**.

Program Interpreter

Helps to bring in dynamic libraries and fix up the relocations. It is used by executables that require **dynamic linking**.

The ELF loader will set up the interpreter's memory image first and jump to the interpreter's entrypoint instead of the main program, before eventually loading the main program.

Essentially, the interpreter has to be loaded before the actual program if dynamic linking is enabled.

System Calls

Allows any application to safely request it through a proper API.

How it works

System calls need to enter and exit Ring 0. This is done using the instructions:

1. SYSENTER (32-bit) or SYSCALL (64-bit)
2. SYSEXIT

SYSENTER **immediately** jumps to a kernel routine to prevent security risks.

Syscall API (user-level)

Sets up registers and invokes the **SYSENTER/SYSCALL instruction.**

Model Specific Register

Special control registers used for many purposes (eg. debugging, execution tracing, etc). You can read and write to them using **rdmsr** and **wrmsr** respectively, which are **privileged instructions**.

During kernel initialization, the location of the syscall entry address in kernel space is written into an MSR. As reading and writing to it are privileged actions, the location of the address is **hidden from the user**.

Kernel Stacks

Kernel requires a stack to execute procedural calls. It also contains **multiple stacks** as the modern kernel is **multithreaded**.

Stacks used by the kernel:

1. Per thread kernel stack
2. Entry trampoline stack
3. Interrupt stack
4. Hard IRQ stack
5. Soft IRQ stack

Per thread kernel stack

It has a size of 16KB. It exists in the **task_struct** of each process and grows towards the **thread_info** structure.

Task State Segment

Essentially holds information about a **task** and is used by the kernel for **task management**.

It also consists of pointers to various stacks:

- 7 interrupt stacks for interrupt service routine usage
- 1 for each protection ring (0, 1, 2)

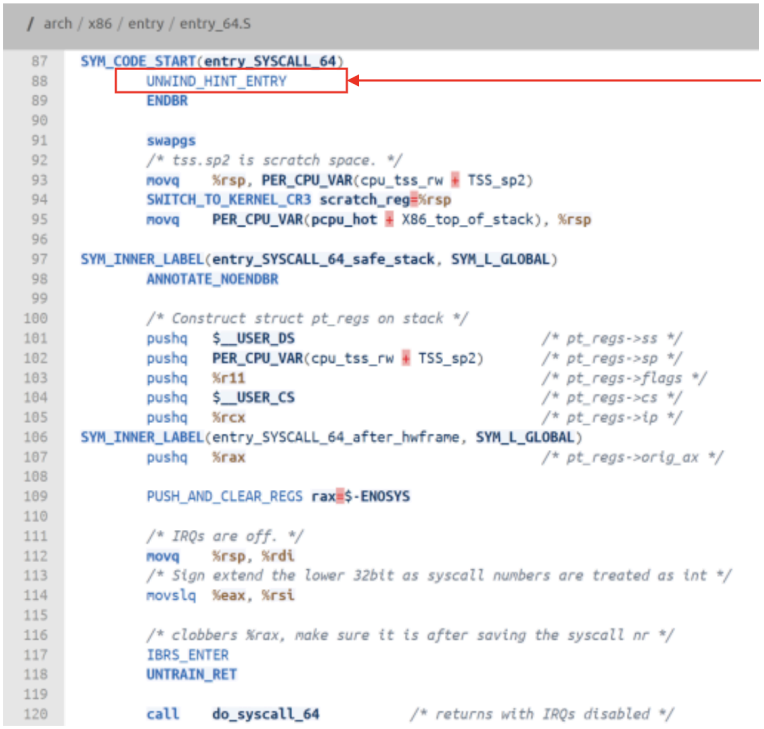
Note: The ring 3 stack is already stored in %rsp.

Each CPU has **only one TSS** and uses them for **ALL tasks**.

Details of SYSCALL

pt_regs is the struct for saving all registers!

Beginning of syscall in kernel space:



/ arch / x86 / entry / entry_64.S

```
87  SYM_CODE_START(entry_SYSCALL_64)
88          UNWIND_HINT_ENTRY
89          ENDBR
90
91          swapgs
92          /* tss.sp2 is scratch space. */
93          movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
94          SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
95          movq    PER_CPU_VAR(pcpu_hot + X86_top_of_stack), %rsp
96
97  SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)
98          ANNOTATE_NOENDBR
99
100         /* Construct struct pt_regs on stack */
101         pushq   $__USER_DS                      /* pt_regs->ss */
102         pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
103         pushq   %r11                            /* pt_regs->flags */
104         pushq   $__USER_CS                      /* pt_regs->cs */
105         pushq   %rcx                            /* pt_regs->ip */
106 SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
107         pushq   %rax                            /* pt_regs->orig_ax */
108
109         PUSH_AND_CLEAR_REGS rax=$-ENOSYS
110
111         /* IRQs are off. */
112         movq    %rsp, %rdi
113         /* Sign extend the lower 32bit as syscall numbers are treated as int */
114         movslq  %eax, %rsi
115
116         /* clobbers %rax, make sure it is after saving the syscall nr */
117         IBRS_ENTER
118         UNTRAIN_RET
119
120         call    do_syscall_64           /* returns with IRQs disabled */
```

FS and GS segment registers

FS points to thread local storage.
GS points to per CPU data structure (user can never see this).

swapgs swaps the values in GS and a “hidden” GS.

Page Table Isolation

A **minimal kernel page table** is stored in user space, just sufficient to transit to the kernel. This helps to protect the kernel space.

At the start of the syscall, the kernel **stores %rsp** into a scratch space and then uses %rsp to swap the user page table with the kernel page table in the CR3 table. The reason %rsp is required is because **a swap requires an extra register**. Note that using the kernel page table requires **flushing the TLB**, which can be time-consuming.

Per CPU Structure

It is a synchronization mechanism that maintains an **array of structure of registers** that have different values depending on the CPU that it is on.

Loading the kernel stack

After that, %rsp loads in the per-thread kernel stack from the per CPU structure.

Saving the registers

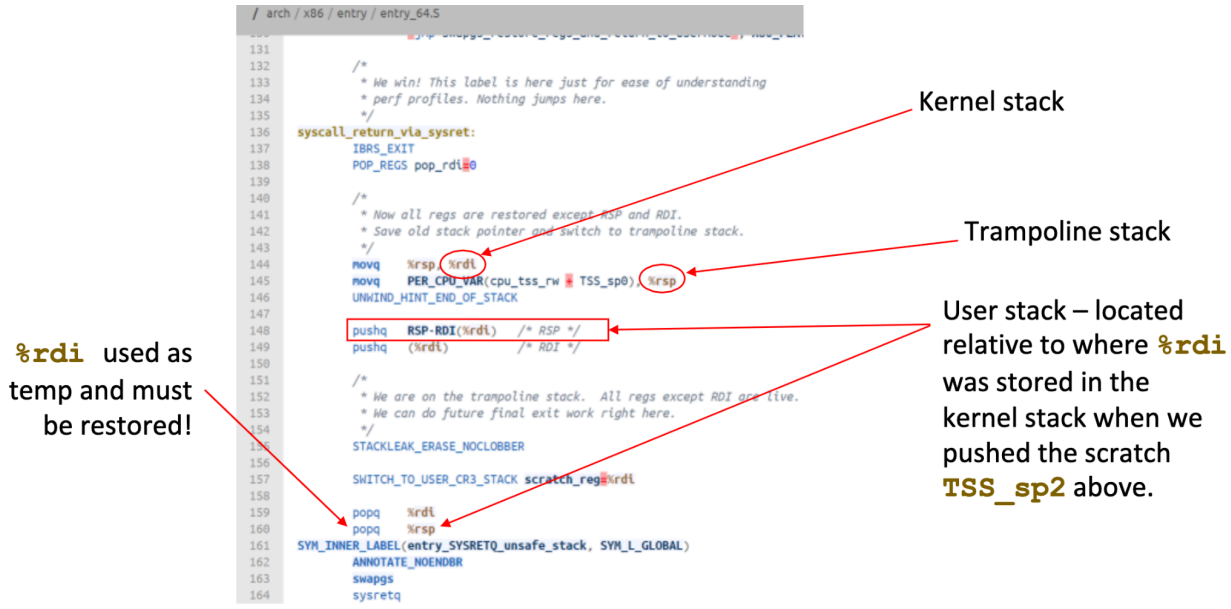
Push all the registers (along with DS, CS, original %rsp from the scratch space) to the top of the current stack, to preserve all the user data. It is saved in the form of a **pt_regs** structure, which also contains the arguments of the syscall.

Finally, the routine is ready to do the syscall.

Doing the syscall

The final syscall entrypoint (**do_syscall_64**) accepts the pt_regs structure along with the syscall number. It accesses the syscall using an array of addresses (known as the **sys_call_table**) and finally calls it with the pt_regs structure.

Returning from the syscall



Basically undos all the actions using **3 stacks**:

1. Pop all registers from pt_regs **except %rdi and %rsp**.
2. Store the **kernel stack into %rdi**.
3. Load the **trampoline stack into %rsp**.
    a. The purpose of this stack is to return the **original %rdi and %rsp to the user stack**.
4. Push the user-values of %rsp and %rdi onto the trampoline stack.
5. Switch back to user stack using %rdi as a scratch register.

6. Pop back the original %rdi and %rsp into their corresponding registers.
    a. This effectively switches from the trampoline to the user stack.
7. Swap back the GS register
8. Jump to the return address.

Intuition
The main reason for the trampoline stack is because transitioning from Ring 3 to 0 allows you access to “extra storage” in the form of the **per CPU structure**. However, transitioning back from Ring 0 to 3 does not provide this extra storage (because you can’t access the structure in user space) unless you use the trampoline stack.

The reason this “extra storage” is needed is to have **an extra register for swapping the page tables**.

Speeding up system calls

To speed up system calls, some of them can be run in user space:
- Linux kernel only maps the pages containing the implementation of the system call.
- The kernel pages are **readable, executable but not writable**.

However, the pages of the syscall are **fixed and known**.

vDSO (Virtual Dynamic Shared Object)

Allows the linker to do **address space randomization** and place the page anywhere in the virtual space for different processes.

Interrupts

Handling interrupts require both **hardware** and **software** cooperation. Interrupts are needed for **responsiveness** of applications and handling unexpected issues (such as errors, faults, exceptions, etc).

Types of interrupts:
1. Asynchronous
    a. Source is external (eg. I/O device)
    b. **Not related to current instruction being executed**.
2. Synchronous (also known as exceptions)

Synchronous interrupts

They can be **processor-detected** (eg. faults, traps, etc) or **programmed** exceptions (eg. syscalls).

Faults

Normal execution cannot continue. Faults can often be fixed. The CPU can just restart its execution **after it's fixed** (assuming it's possible). Otherwise, it needs to terminate gracefully.

Due to deep pipelines and superscalar executions in modern processors, a **hardware replay queue** is used for faulting instructions to be replayed.

Trap

Deliberately set and triggered by an exception in a **user process**.

It is used by debuggers. The parent (debugger) will replace certain bytes (breakpoints) in the memory of the child process (the debugged process) with INT 3. When the child process hits the address, it will be trapped to the parent. The parent will be able to inspect registers, memory, etc. Then, it will replace INT 3 with the original byte and resume execution of the child process.

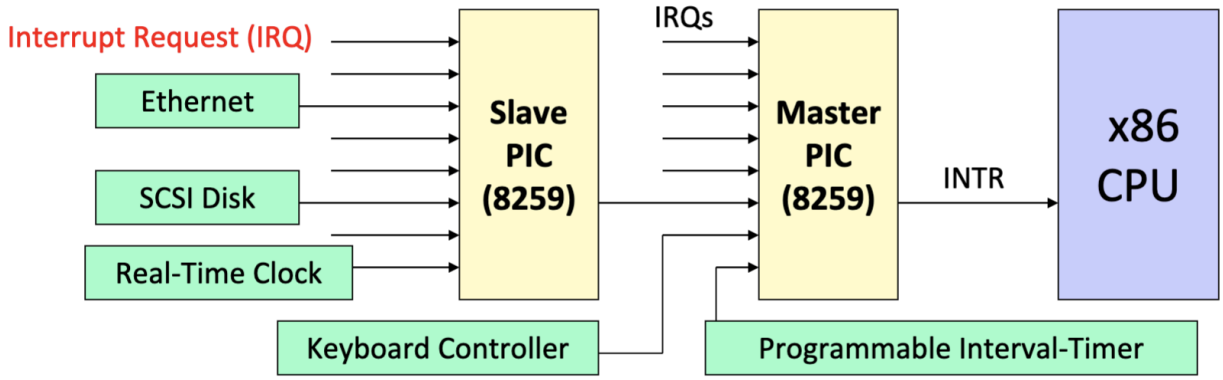
Error Exceptions

Usually most exceptions (such as divide by zero, privileged instruction, etc) are sent back as **signals** to the user process for handling.

This does not apply to **page faults**. The kernel must handle it.

Programmable Interrupt Controller

Single Processor System



Legacy PC Design

Flow

1. I/O Devices send IRQs to the PIC.
2. PIC will resolve the interrupt to be handled by **checking its priority and if it's masked**.
3. PIC will interact with the CPU:
    a. PIC sends INT signal to the processor.
    b. Processor sends back three signals for the PIC to respond:
        i. X86 CALL opcode
        ii. Low byte of call address
        iii. High byte of call address
    c. Processor executes the CALL instruction using the given address (interrupt service routine).

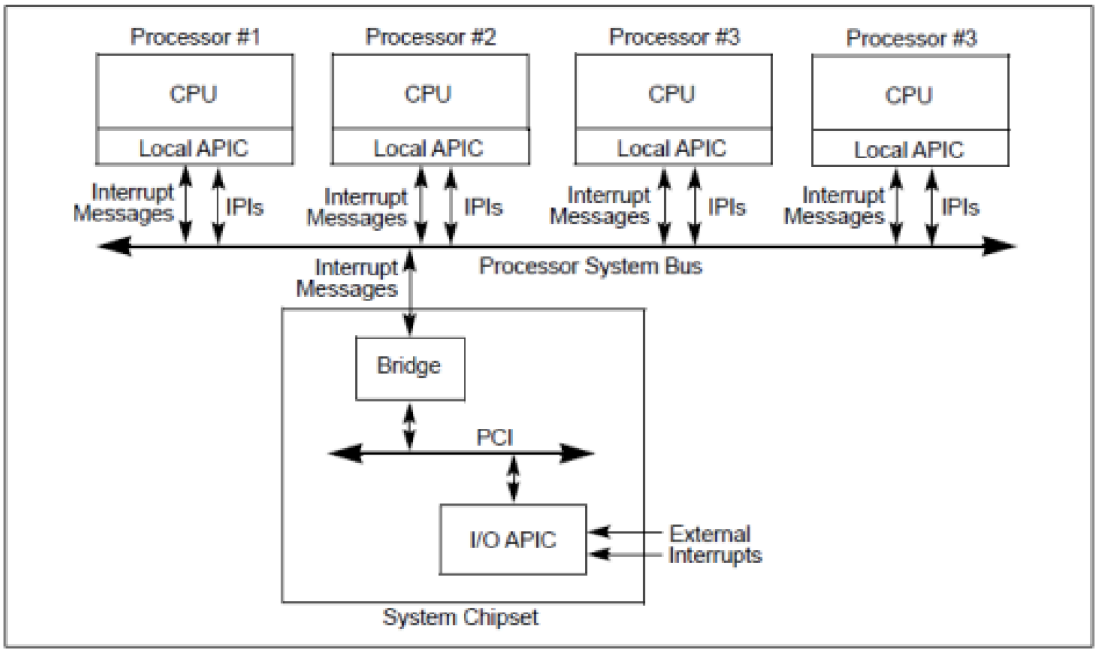
Disadvantages

It cannot handle **multiprocessor systems**:

- Cannot resolve which **CPU to send the interrupt to**.
- Cannot implement **inter-processor interrupts** (when processors interrupt each other).

Multi Processor System (APIC)

Modern multiprocessor systems use the **Advanced Programmable Interrupt Controller**.



Each CPU consists of a core and a **local APIC**. It also contains a **local vector table**. There is also a separate **I/O APIC** (can be more than one).

Interrupts are routed to the CPU over the **system bus**. Inter-processor interrupts are also supported.

Local Vector Table

Defines the vectors for the interrupt service routines.

Masking Interrupts

Most interrupts can be “masked” by clearing the **IF flag** in the EFLAGS register for a core.

Assigning IRQs to Devices

IRQs are usually assigned **on boot** by the PCI bus. Some IRQs are fixed (eg. interval timer). Linux device drivers request IRQs when the **device is opened**.

Two devices that aren’t used at the same time can share an IRQ.

Interrupt Vector

An interrupt vector is an index into the **interrupt descriptor table**.

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20	#VE	Virtualization Exception	EPT violations[6]
21-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Intel Reserved

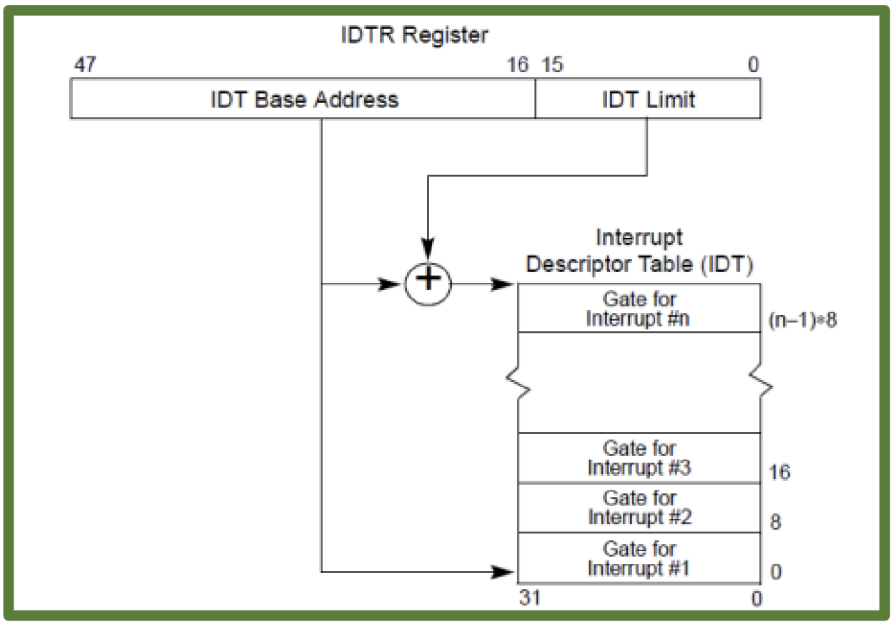
The first 32 are already reserved for non-maskable interrupts and exceptions. The rest can be assigned as needed.

Vector 128 is used for the Linux syscall.

Interrupt Handling (by x86)

The **Interrupt Descriptor Table Register** (IDTR) points to an **Interrupt Descriptor Table** (IDT).

Interrupt Descriptor Table



It consists of either:

1. Task-gate descriptor: Meant for the “double fault” exception. This is caused by kernel bugs and will cause the processor to shut down.
2. Interrupt-gate descriptor: Disables further interrupts
3. Trap-gate descriptor: Further interrupts still allowed

Gate Descriptor

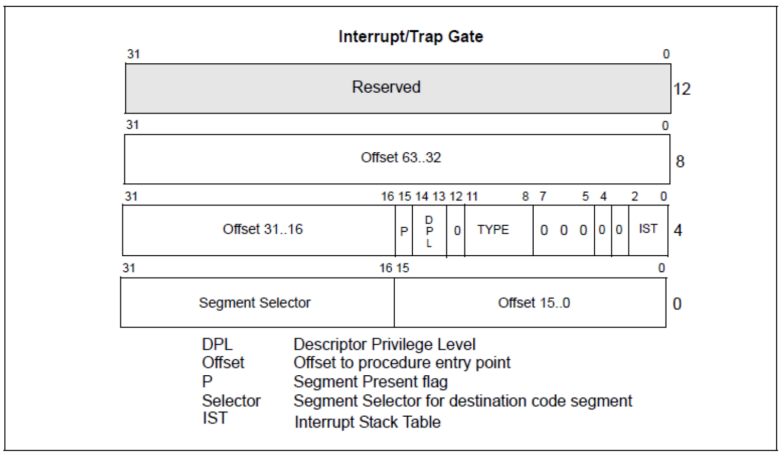
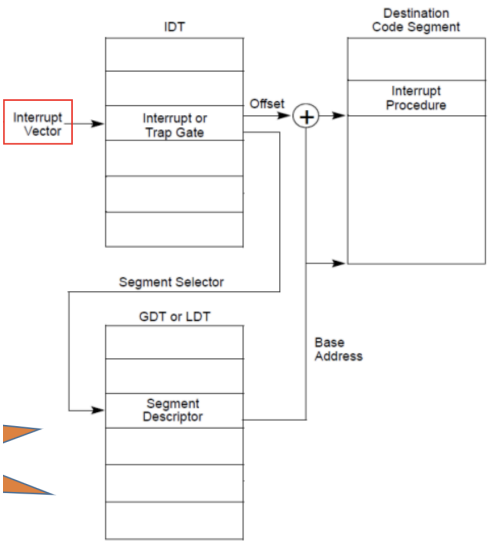


Figure 6-8. 64-Bit IDT Gate Descriptors



Note that **IST** indicates which stack to use:

- If privilege level is **sufficient**, hardware sets the new stack to the IST value. Otherwise (eg. coming from user mode), it will first set it to the **per-thread kernel stack**. This is done using the same entry steps as SYSCALL:
  - **swapgs** to bring in per CPU data structure.
  - Switch to the **full kernel page tables**.
  - Change stack to per-thread kernel stack.
  - **Finally, switch to the IST**.
- If IST is zero, use the corresponding ring stacks specified by the TSS (eg. RSP0, RSP1, RSP2).
- Each stack is of size 4K (1 page).

Summary (for x86)

1. IRQ is sent via the APIC system and interrupts the processor.
2. The **interrupt vector** is used as an index into the **interrupt descriptor table** to allow the CPU to jump to the **interrupt service routine**.
3. The corresponding **stack switching** occurs based on the IST and DPL.
   a. In 32-bit, this is done by hardware.

    b. In 64-bit, this is done by entry_SYSCALL_64.
4. (This only applies to an OS that supports segmentation)
   Base address is determined from the GDT/LDT using the segment selector in the gate descriptor.

Initialization during the boot process

- The interrupt descriptor table is first initialized by the **BIOS**.
- When Linux boots, it will be overwritten by a new table (known as **idt_table**).
- **Drivers will request for IRQ** from the kernel.
- Table will be updated again **after the 7 ISTs are set up**.

Nested Interrupts

Linux used to support nested interrupts but it was removed to avoid complex solutions.

However, it is still possible to have nesting between **exceptions and interrupts** (but it is fairly restrictive):

- An exception (eg. page fault, system call) cannot preempt an interrupt.
- An interrupt can preempt an exception

Interrupt Handling (by Linux OS)

This section can be divided into two parts.

Top Half (immediate handler)

Known as the **interrupt service routine**. It does the absolute minimum work to handle the interrupt and **return as quickly as possible**. This is because **interrupts will be disabled** (as some routines may be **critical**), which may increase the latency for handling other interrupts.

This means that you cannot:

- Sleep or **call something that might sleep**.
- Cannot refer to **current** (pointer to the current process).
- Cannot call schedule.
- Cannot call **down()** on a semaphore.
- Cannot transfer data to/from user space.

The rest of the **non-critical processing** is pushed to the bottom half.

Interrupt Stacks:

1. Exception stack (used by exceptions **per process**)
2. Hard IRQ stack (used by interrupts **per processor**)

3. Soft IRQ stack (in 64-bit, it is equivalent to the **Hard IRQ stack**)

There are three types of interrupts:

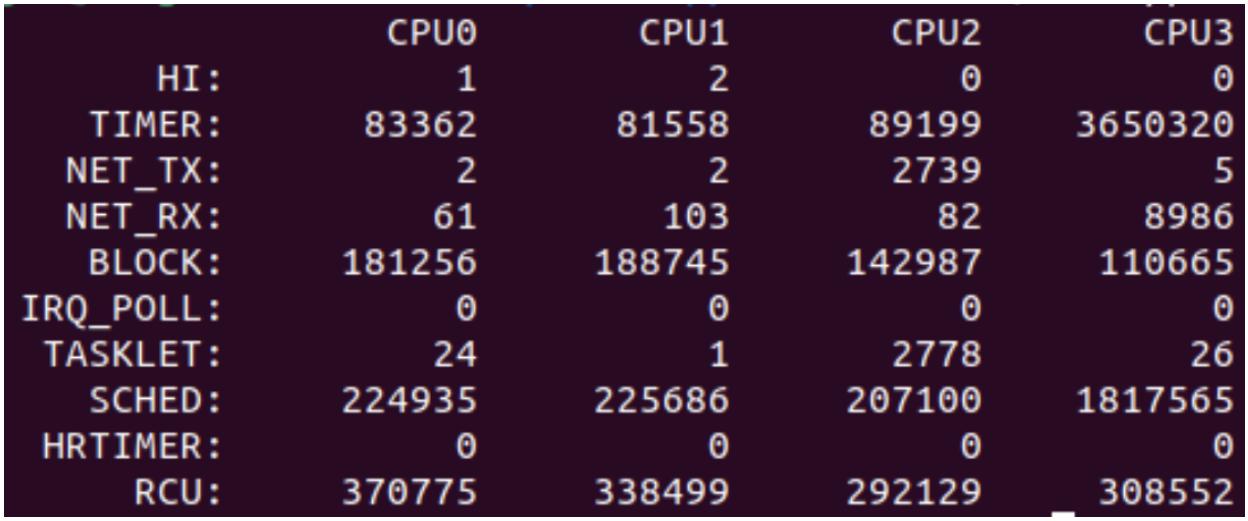
1. I/O interrupts
2. Timer interrupts
3. Interprocessor interrupts

Bottom Half (deferrable functions)

SoftIRQ

Also known as “software” IRQ. Uses **softirq_vec[]**. Can be interrupted and concurrent (needs to be protected through spinlocks).

Types of SoftIRQs:



	CPU0	CPU1	CPU2	CPU3
HI:	1	2	0	0
TIMER:	83362	81558	89199	3650320
NET_TX:	2	2	2739	5
NET_RX:	61	103	82	8986
BLOCK:	181256	188745	142987	110665
IRQ_POLL:	0	0	0	0
TASKLET:	24	1	2778	26
SCHED:	224935	225686	207100	1817565
HRTIMER:	0	0	0	0
RCU:	370775	338499	292129	308552

Ksoftirqd

This is a daemon that runs on each processor that constantly checks for softIRQs.

```
for(;;) {
    set_current_state(TASK_INTERRUPTIBLE );
    schedule( );
    /* now in TASK_RUNNING state */
    while (local_softirq_pending( )) {
        preempt_disable();
        do_softirq( );
        preempt_enable();
        cond_resched( );
    }
}
```

Tasklets

Can be **statically or dynamically allocated**, unlike SoftIRQ which is fixed. They are also **not re-entrant**. This means tasks of the **same type** can only run serially in the entire system.

Work Queues

Work queue functions run in the **process context** (has its own kernel stack), unlike SoftIRQs and tasklets which run in the **interrupt context**. Hence, they are allowed to sleep. They are run by a worker thread which waits for work.

Kernel Threads

Each bottom half will have its own context, unlike work queues. A new kernel thread is spawned for each interrupt.

Summary

	ISR	SoftIRQ	Tasklet	WorkQueue	KThread
Will disable all interrupts?	Briefly	No	No	No	No
Will disable other instances of self?	Yes	Yes	No	No	No
Higher priority than regular scheduled tasks?	Yes	Yes*	Yes*	No	No
Will be run on same processor as ISR?	N/A	Yes	Yes	Yes	Maybe
More than one run can on same CPU?	No	No	No	Yes	Yes
Same one can run on multiple CPUs?	Yes	Yes	No	Yes	Yes
Full context switch?	No	No	No	Yes	Yes
Can sleep? (Has own kernel stack)	No	No	No	Yes	Yes
Can access user space?	No	No	No	No	No

*Within limits, can be run by ksoftirqd

Signals

It is a **notification** of an event:

- User application stops immediately.
- User-supplied **signal handler** will execute to completion.
    - If a handler is not supplied, default action is taken (usually a core dump and process termination).
- User application resumes.

Can be sent using the **kill** command.

The **kernel** needs to be involved in user-level signals because:

- Need to interrupt the user process and resume it.
- Signals may arise from exceptions.
- IPC requires OS checking and intervention.

There may be **race conditions** in multicore systems as signals have **data structures such as queues** (for pending signals).

Signal Generation

Kernel/User process can send a signal to a user process. The pending signal list of the receiving process is updated.

If it is sleeping, it needs to be woken up. Otherwise, we need to do an **interprocessor interrupt**.

Signal Delivery

Before an **interrupt routine** (in kernel mode) returns to user mode, a pending signal is checked. Then, the signal procedure is invoked for the receiving process.

Processes

Modern Linux refers to both processes and threads as **tasks**. Tasks are **scheduling entities** in Linux.

Task components	Description
task_struct	Known as the **process control block**. It holds all information about a process/task.
thread_info	Located with the per thread kernel stack. It contains architecture dependent info and **status flags**.
thread_struct	Located at the end of task_struct. Holds architecture dependent info such as register states, etc.

Task States

- **`TASK_RUNNING`** – task is either executing on a CPU or is awaiting execution
- **`TASK_INTERRUPTIBLE`** – task is sleeping but can be awaken by an interrupt
- **`TASK_UNINTERRUPTIBLE`** – task is sleeping. Any interrupt delivered to it will not change its state.
  - Seldom used but needed for certain critical wait situations
- **`TASK_STOPPED`** – task is stopped
- **`TASK_TRACED`** – task is stopped by a debugger (using **`ptrace`**)

Zombie process - A process which is no longer running and **awaiting termination**.
Orphan process - A process whose parent died. It is automatically adopted by init.

Process Creation

Copy-on-Write is used to reduce the time taken for process creation. Allows the child to share part of its context, such as **memory space, file descriptors and signal handlers**. This is done using the **clone()** system call.

vfork() is similar to clone, except that the **parent will block** until the child exits or performs **execve()**.

Process Switching

Done using the **schedule()** function. Need to switch the **page global directory, kernel mode stack and the hardware context** using the **switch_to()** macro.

Timekeeping

Usage of multiple pieces of hardware such as the **real-time clock**, **CPU local timer**, etc. Timekeeping is important to find out the time and date, determine the amount of time a process is executing, etc.

For uniprocessors, time-keeping activities are triggered by interrupts raised by the **global timer**. For multiprocessors, it is mostly the same, but CPU-specific activities are raised by **local APIC timer**.

jiffies - counter that stores **number of elapsed ticks** since the system was started.
xtime_sec - Stores the current time and date in seconds.

Process Scheduling

Need to:

1. Minimize response time
2. Maximize throughput (jobs per second)
3. Fairness

Turnaround time is the time elapsed from submission to completion.
Response time is the average time elapsed from submission to the **first response produced**.
Waiting time is the time elapsed from arrival to the first moment it executes on the CPU.
Completion time is the time taken for a process to wait and finish executing.

A **global clock** is used for parallel processes.

FCFS

One program runs non-preemptively until it is finished or blocked. Performance is highly dependent on the **order in which the jobs arrive**.

Round Robin

Each process runs for time quantum. Need to ensure **time quantum >> context switch time**.

It is better for **short jobs**.

Note that **cache state** must be shared for RR, but that is not true for FIFO. Hence, even without context switch time, there is still implicit overhead!

SJF / SRTF

Provably optimal for their non-preemptive and preemptive types of algorithms. However, note that this can **lead to starvation** if many small jobs keep coming and block the large jobs.

Prediction of compute time

This is done by using adaptive policies, such as using an exponential moving average of previous CPU bursts.

Priority Scheduling

Use a **priority number** to associate with each process. We can increase this priority gradually for processes that have not executed in awhile. This is calculated using **niceness** in Linux, with -20 being the most favorable and 19 being the least favorable.

Multilevel Feedback Queue

Multiple queues with different priorities, each with its own scheduling algorithm (eg. RR, FCFS). If quantum expires, drop the level. Otherwise, raise the level.

Long-term scheduler - Decides processes to put into the ready queue.
Short-term scheduler - Decides processes to execute from the ready queue.

Lottery Scheduling

Give each job a certain number of lottery tickets, and pick a winning ticket **at each time slice**. The advantage is that adding/deleting jobs will **affect all jobs proportionally**.

Data Structures

Each CPU has a **runqueue**, with three subqueues. Each task has a **processor affinity mask** which tells which CPU the task can run on.

Types of Schedulers

Real-time Scheduler

A real-time task can only be replaced by another task:

- If it has a lower priority.
- If it blocks, dies or voluntarily relinquishes the CPU.
- The scheduler is round robin.

O(n) Scheduler

Time is divided into **epochs**. Every task can execute up to its time slice in the **current epoch**. Half of the unused time is added to the next epoch. Each task has a **goodness** factor that is a function of the remaining time and the priority.

This requires a loop over all tasks at every context switch.

O(1) Scheduler

Maintains an **active and expired runqueue** per CPU. When the active array is empty, swap the two arrays. The array contains a doubly linked list **for each priority value**.

A **priority bitmap** is used to check if a task for a priority level has **at least one task in it**.

Earliest Eligible Virtual Deadline First Scheduling (EEVDF)

Dynamically handles fairness. The set of runnable tasks may change and/or the weight of the tasks may change. Hence, we use **virtual time**.

Virtual runtime is calculated using the actual time and the task's **load weight**. The loadweight translates niceness to weights.

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

All tasks will get the same **virtual time slice** to run. If nice < 0, virtual runtime grows **slower** than real time. If nice > 0, virtual runtime grows **faster** than real time.

The scheduler uses an **augmented RB tree** that is sorted by **vruntime**. A task is **eligible** if lag >= 0, which is the difference between the weighted average runtime of all tasks and the actual time the current task ran.

Hence, if a task is not eligible, continue traversing **left**. If a task is eligible, then traverse downwards to find the node with the **minimum deadline** (contained as extra augmented data in the node).

Updating of deadline for a task:

- Obtain a virtualized timeslice, s, that is:

$$s = \frac{1024}{w_i} * \text{base\_timeslice}$$

- Then:
  - If new task, then deadline = $\frac{s}{2} + v_i$
  - Otherwise, deadline = $s + v_i$

Completely Fair Scheduler (CFS)

Unlike EEVDF, CFS only cares about the weight parameters, which results in reduced fairness and performance.

Group scheduling - Time is divided among **scheduling entities**, not the tasks directly.
Bandwidth control - Allows users to specify a limit to CPU time for a group or process hierarchy.
SMP Load Balancing - Each CPU has its own runqueue, which can be balanced between each other.
Scheduling Domain - Used for NUMA architectures to prevent tasks from being scheduled “all over”.

Memory Management

All processes share physical address space. User programs run in a standardized virtual address space.

Page tables can be very huge, hence multilevel page tables are used which can be dynamically allocated.

Translation from VA to PA can result in one more memory access. This is resolved by the **TLB** to cache recently used address mappings. TLBs are also faster than caches.

The root page table is pointed to by a field in the **CR3 register**.

Process Context ID

Used to distinguish one process’ address space from another, helping in faster context switching as sometimes you don’t have to flush the TLB.

Canonical Addresses

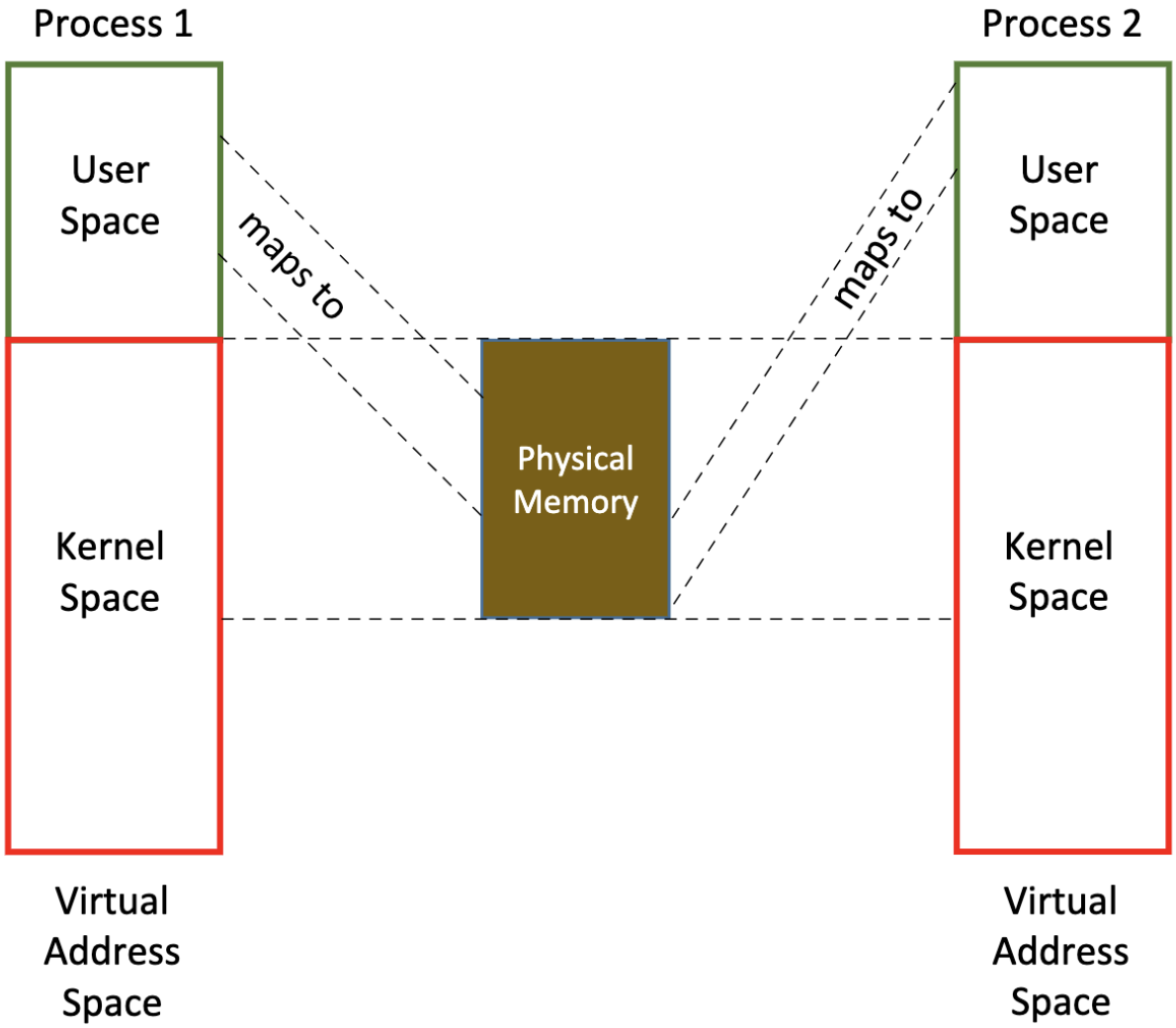
A linear/virtual address is in canonical form if address bits 63 to the most-significant implemented bit are either all ones of all zeros.

Reserved Page Frames

Pages contained in a reserved page frame can never be dynamically assigned or swapped to disk. They include frames outside the available physical address range and frames containing kernel code and initialized data structures.

Linux Memory Map

It should be noted that **ALL** physical memory maps to a fixed region in kernel space. This means that a fixed region in kernel space is mirroring the actual physical memory.



The kernel uses a single set of page tables. It is also the page table for process 0. Each process also has its own page table in the kernel to guarantee process isolation.

Page Table Isolation

Two sets of page tables are maintained for each process. Page tables are switched whenever the kernel is entered via syscalls. This is meant to protect the kernel address space and user space from accessing each other unintentionally.

Virtual Memory Area

Keeps track of a process's memory mapping, such as the start and end addresses of a memory region. Heap area virtual pages are allocated using **brk()**. Files can be **mmap()**-ed into the virtual address space.

Lazy expansion

Although memory can be allocated using **brk()**, memory does not actually get allocated until the kernel actually assigns page frames for it. This is known as **lazy expansion**. If the memory is accessed when there is no actual memory allocated, page fault occurs as there is no PTE.

A frequent operation is to find the region a virtual address is in. This can be done by traversing the linked list of VMAs linearly or using a red-black tree (for processes with many memory regions).

Physical Memory

Physical memory is divided into memory zones. One primary reason for this is for **direct-memory access**.

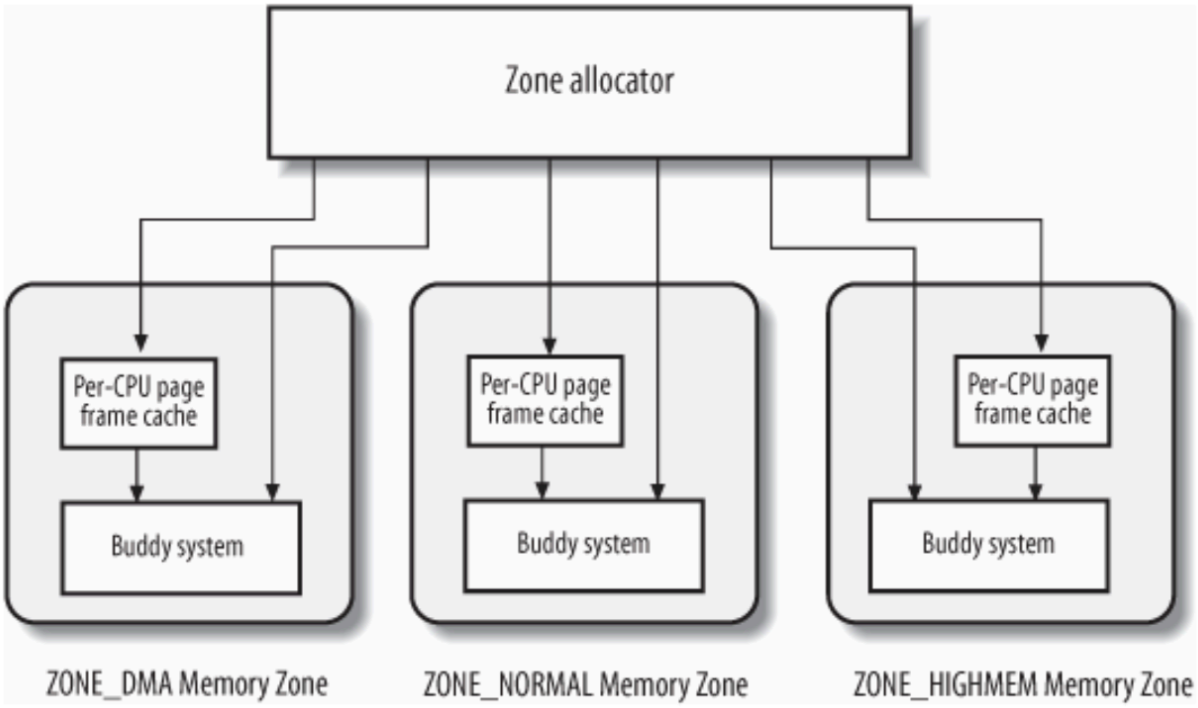
Direct Memory Access

It allows I/O devices to access memory without involvement of the CPU. Used by many devices such as disk drive controllers, graphics cards, network cards and sound cards.

However, DMA only works with **physical addresses** and must be set up by the OS.

Memory Zones

Portions of physical memory are divided into memory zones. Page allocation must be done to the correct zone. This is done using the **zoned page frame allocator**.



Zoned Page Frame Allocator

Zone allocator receives requests for allocation and deallocation of dynamic memory. This is done using the **Buddy System Algorithm**. A **small cache** is also used for fast, single frame allocation requests to improve performance.

If there is not enough free memory, a request must be **put on hold**. However, some requests should not be blocked (especially in ISRs).

One solution is to have a reserved page frame pool for allocation, which can be “topped up” if it is too low.

kmalloc() is used to allocate contiguous **physical** memory, unlike vmalloc().

Per-CPU Page Frame Cache

For each CPU, pre-allocate some page frames to be used by that CPU. Consists of **hot cache** for the CPU to write immediately to that frame and **cold cache** which is used for DMA.

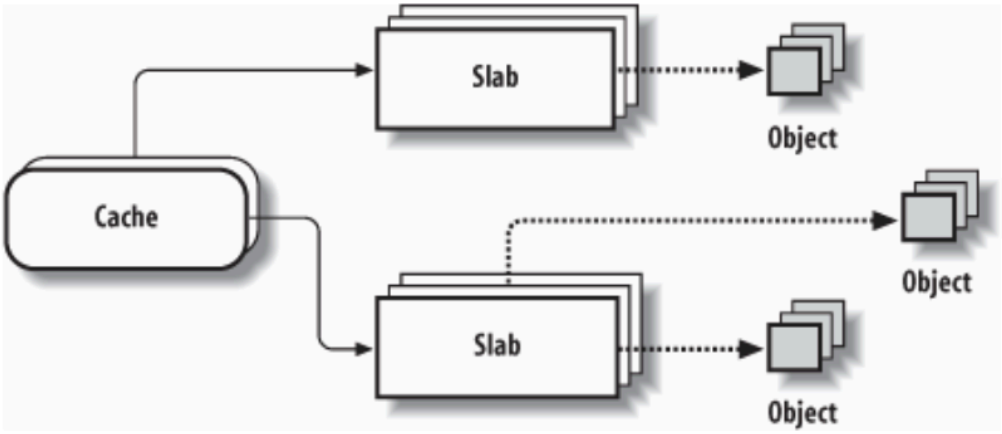
SLAB Allocator

Based on the idea of caches, each cache stores a collection of slabs, which contains objects of the same type (in a cache). It keeps **commonly used objects** in an initialized state available for

use by the kernel. Without it, the kernel will spend much of its time allocating, initializing and freeing the same object.

It fulfills 3 principle aims:

1. Eliminate internal fragmentation which would otherwise be caused by the buddy system.
2. Caching of commonly used objects (as the slabs have already initialized the objects).
3. Better utilization of **hardware cache**. This is done by **padding** slabs in the same cache with a “color”, which prevents cache thrashing between slabs.



Hierarchy

- Each cache contains slabs.
- Each slab contains contiguous pages.
- Each page will contain adjacent objects (no internal fragmentation).

If not enough slabs, require the **buddy system allocator** to allocate a new slab!

SLUB Allocator - For NUMA architectures, it considers local slabs and remote slabs!

Page Fault Handler

Minor fault: Do not need to block the current process (page is in memory but not loaded in MMU).
Major fault: Must put the current process to sleep.

Page Frame Reclamation Algorithm

There are 4 types of page frames:

1. Unreclaimable (eg. Reserved pages)
2. Swappable (eg. Anonymous pages, shared memory)

3. Syncable (eg. Normal mapped pages in the address space)
4. Discardable (eg. Unused pages included in memory caches such as slab allocator)

A page is **mapped** if it is **part of a file**. Changes must be saved to the corresponding file.

A page is **anonymous** if it is not mapped. It must be saved to a dedicated disk partition known as a **swap area**.

General Principles of PFRA:

- Free “harmless” pages such as infrequently used pages of various caches in the kernel.
- Make all user mode pages reclaimable.
- Reclaim a shared page frame by unmapping **all PTEs referencing it in a single go**.
- Reclaim “unused” pages only.

Reverse Mapping

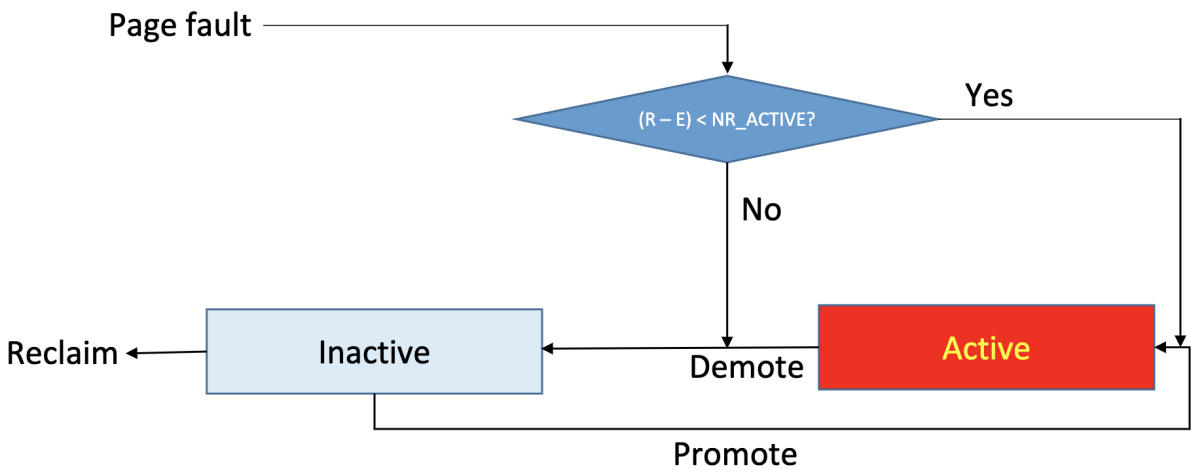
Every page frame has a corresponding **page descriptor**. This page descriptor stores the **_mapcount** and **mapping**. They store the number of PTEs pointing to it, where the page belongs to (eg. swap cache, normal address space) and the list of PTEs referencing it. The list can be handled by a double linked list or an **interval tree**.

Page Reclamation

Also known as **demand paging**. For each memory zone, 2 LRU lists are maintained. The aim is to capture the working set of a process.

Freshly faulted pages are at the head of the inactive list (pushing the current tail out of memory), while the pages that are frequently accessed are promoted to the active list.

A fresh page can be immediately placed in the active list, **if (R - E) < NR_active**. In other words, if that page was previously placed in the active list, it would not have been faulted. Hence, now that it is back again, you put it into the active list to make up for it. This allows you to estimate the **working set** of a process.



To implement the above, you would maintain a counter for inactive evictions and activations for each zone. On eviction of a page, it is first stored in a **page cache radix tree**. If a freshly faulted page exists within this tree and there is an eligible refault distance, it will immediately be activated.

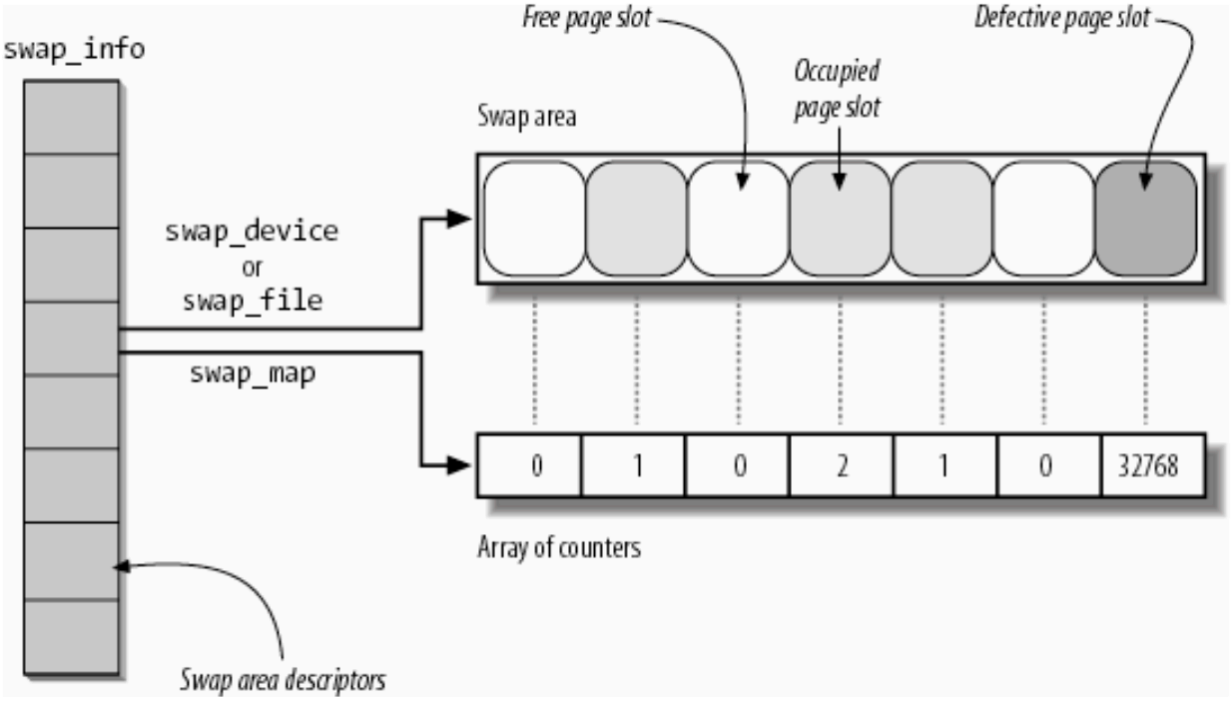
Page Cache

It is the “bridge” between memory and disks. It acts as the cache for the disk. If there is low space in the page cache, eviction is triggered.

A **backing device** is a block device whose content may be cached in memory. It has a dedicated kernel thread for flushing.

Swapping

A **swap area** is a disk partition or a file included in a larger partition. It consists of **page slots** and organized into **swap extents**.



There may be **race conditions** during swapping. A solution would be using a **swap cache**. It is basically a locking mechanism.

Swapping may also be **batched** because of high disk access overhead.

zswap

It is a cache for swapped pages that compresses and stores swapped pages into a memory pool rather than writing back to disk (single memory is getting big these days).

Synchronization

Kernel Control Paths

- Interrupt Handlers
- Exception Handlers
- User-space threads in kernel (system calls)
- Kernel threads (idle, work queue, pdflush, etc)
- Bottom halves

Per-CPU variables

They are used to reduce the need to synchronize across CPUs. Each processor core will have its own copy of the variable, hence **no data race**! There are also **performance benefits** as bouncing cache lines can be avoided.

Atomic Operators

Simply indivisible primitive operations that are implemented via the instruction set of the underlying CPU architecture.

Barriers

Prevents reordering of instructions on the compiler (optimization barrier) and hardware (memory barrier). Instructions before a barrier must be in order before proceeding on.

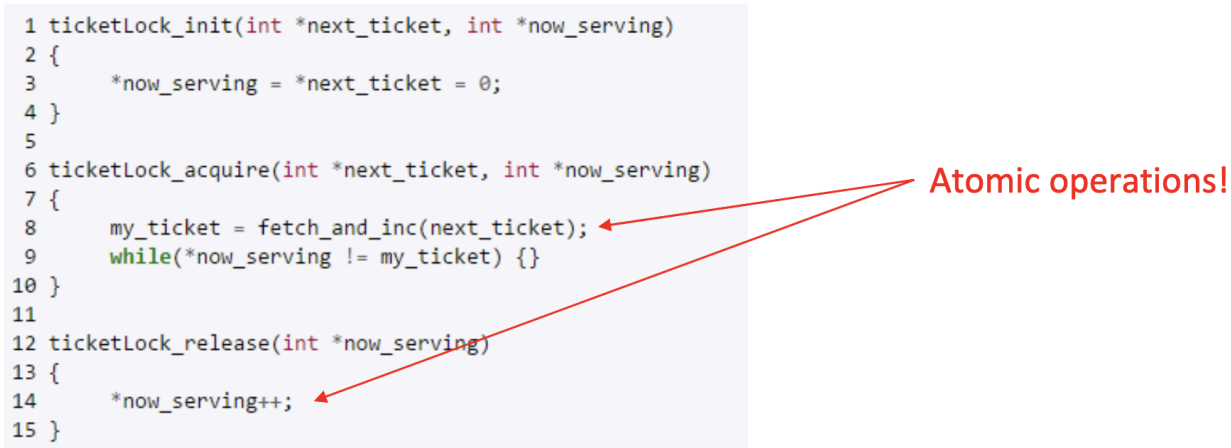
Spin Lock

Used to implement critical sections. **Never block while holding a spinlock!**

Ticket Lock

Ensures FIFO fairness for locking. This used to be implemented in Linux.

```
1 ticketLock_init(int *next_ticket, int *now_serving)
2 {
3       *now_serving = *next_ticket = 0;
4 }
5
6 ticketLock_acquire(int *next_ticket, int *now_serving)
7 {
8       my_ticket = fetch_and_inc(next_ticket);
9       while(*now_serving != my_ticket) {}
10 }
11
12 ticketLock_release(int *now_serving)
13 {
14      *now_serving++;
15 }
```



If a spinlock holder is interrupted, there may be:

1. Delay for holder and delay for every other thread waiting on the lock.
2. Interrupt handler may access data protected by spinlock.

This can be solved by disabling interrupts before acquiring spinlocks (for non-interrupt contexts). This can be done using **spin_lock_irqsave()** and **spin_unlock_irqrestore()**.

Semaphores

Read CS3211.

Reader-write Locks

Implemented to handle readers and writers.

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;


read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);


write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_rwlock);
```

Seqlock

It is similar to read-write locks but **writers get much higher priority**.

Reader logic: Only proceed if sequences match **and** are even.

```
do {
    seq = read_seqbegin(&seqlock);
              ...
} while (read_seqretry(&seqlock, seq));
```

Writer logic: Acquire lock and increase sequence field. It will increase once more when releasing the lock.

Big Kernel Lock

It used to be the only SMP lock in the kernel.

Read-Copy Update

Allows many readers and many writers to proceed concurrently. It is lock free and can only be implemented by data structures that are **dynamically allocated** and **referenced by means of pointers**.

Writers make a copy of the data structure by dereferencing the pointer. When a writer is done, it changes the pointer to the data structure **atomically**.

To handle **memory leaks**, the kernel requires every potential reader to execute **rcu_read_unlock**, this enables the kernel to keep track of the number of references.

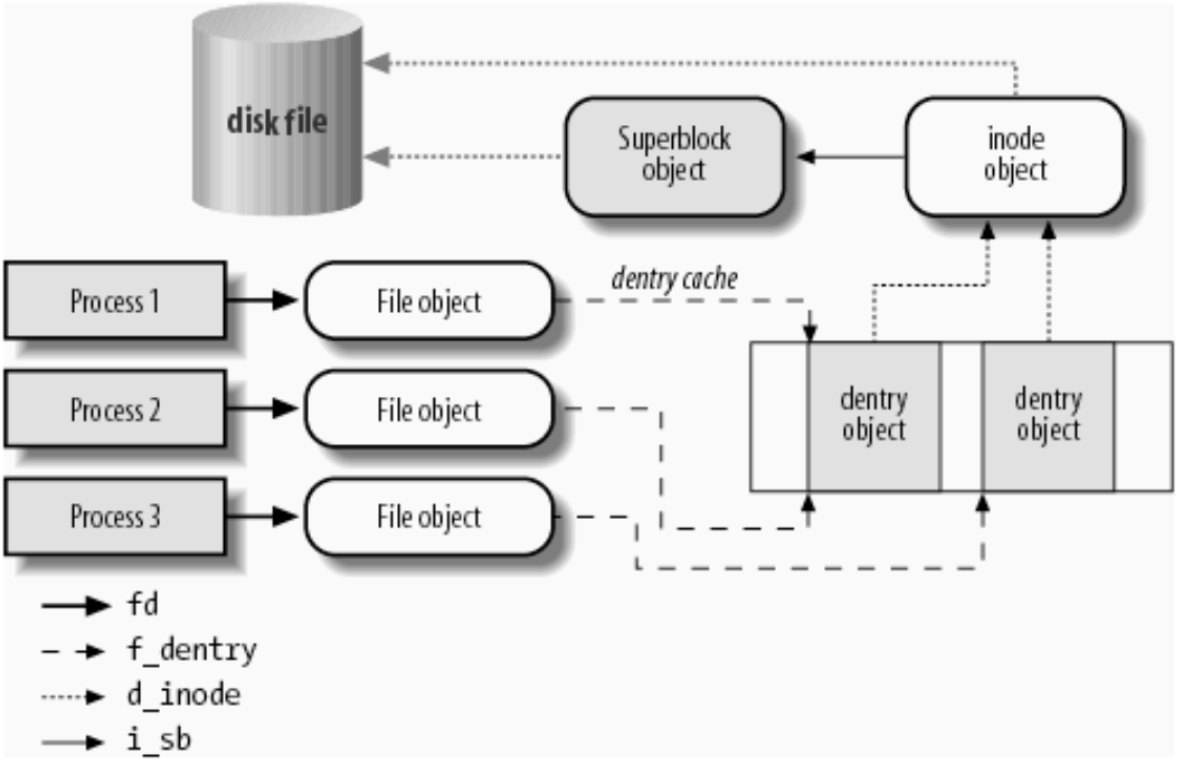
File System

Virtual File System

The VFS is a uniform API to abstract away the complexities of more concrete filesystems, such as ext2. For example, device files exist in the system directory but are intrinsically different from normal files, but VFS hides the difference.

Common File Model

It is a model capable of representing **all supported filesystems** and any physical representation must be translated into this model.



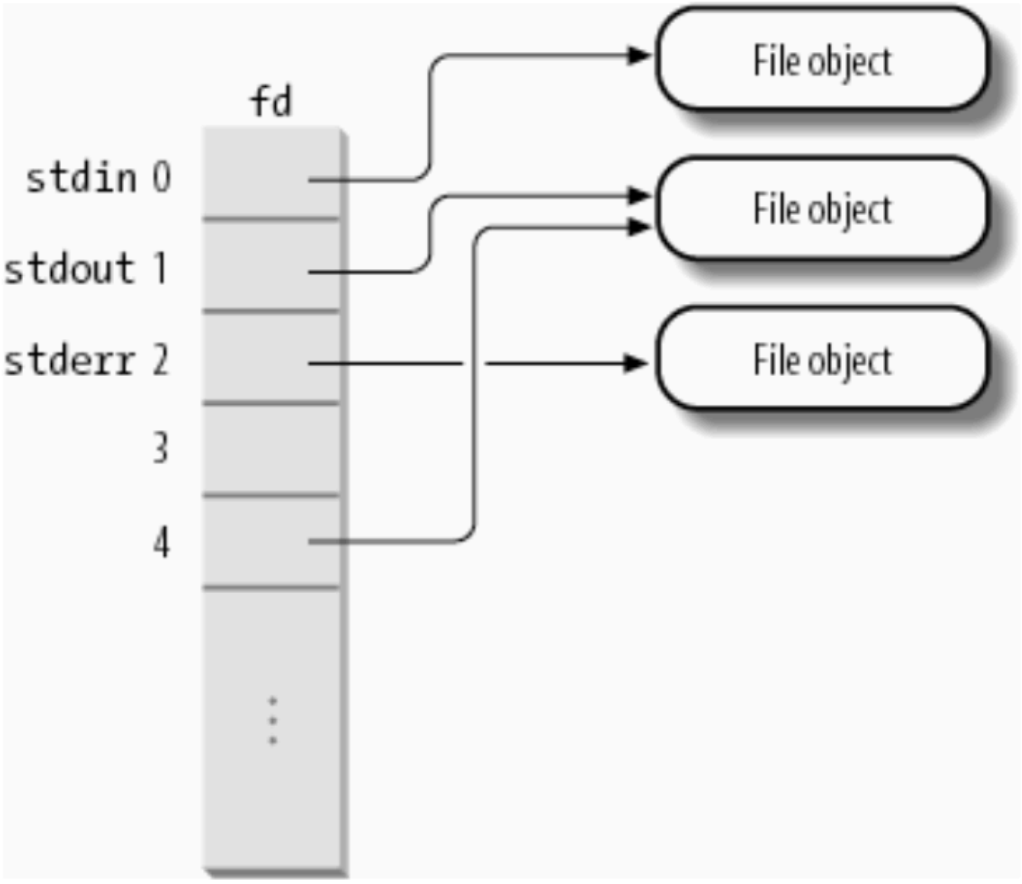
Object	Description	Implementation
superblock	Stores information about a mounted filesystem. Corresponds to a **filesystem control block** stored on disk.	All superblock objects are linked in a circular doubly linked list.
inode	Stores general information about a **specific file**. It contains an **inode number** that uniquely identifies the file within the filesystem. There is a **VFS inode** and a **disk inode**.	The VFS inode is a struct member of the disk inode. It acts as an inode cache. The VFS inode also contains a **set of function pointers for performing file and inode operations**. This allows for implementation of virtual filesystems like **/proc**.
file	Stores information about the **interaction between an opened file and a process**. Only exists in kernel memory while a file is being opened by a process. The main information it contains is the **file pointer**, which is the current offset in the file for the next operation to take place.	It is allocated via a **slab cache** known as **filp**.

dentry	Stores information about the **linking of a directory entry** with the corresponding file. This allows remembering the resolution of a given file without having to search the filesystem.	They are only stored **in memory** and allocated via a slab cache known as **dentry_cache**. Constructing a dentry object takes time and there is usually temporal locality in file usage, hence the cache is used. This cache can also act as the inode cache as inodes and dentry-s are associated.

Files associated with a process

Each process descriptor has a pointer to its **fs_struct**. This is the interpretation of pathnames referred to by a process.

The **files** field of a process descriptor tells you which files are currently opened by a process. This field contains the **fd** field. It is basically the local file descriptor table.



Namespace

Each process might have its own **tree of mounted filesystems** (known as the namespace). However, usually processes just share the same mounted filesystem.

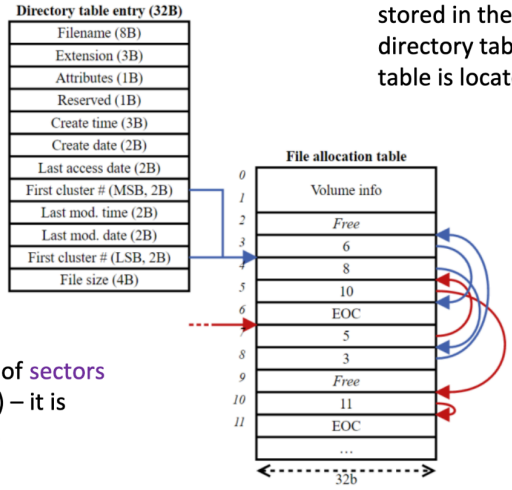
Note that you can mount the same filesystem multiple times, but they all share only one superblock object.

File System Formats

File Allocation Table

Typically used in many embedded media.

A directory table (a.k.a. a folder) is a special file stored in the data area that contains a number of directory table entries. A special root directory table is located in the start of the data area.

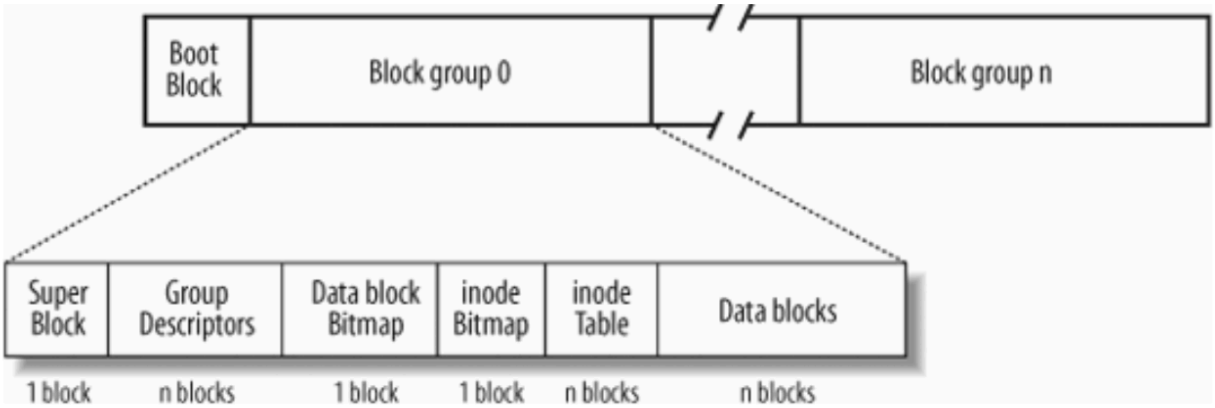


A cluster consists of a number of sectors (the storage unit of the device) – it is predefined at formatting time.

Extended Filesystem

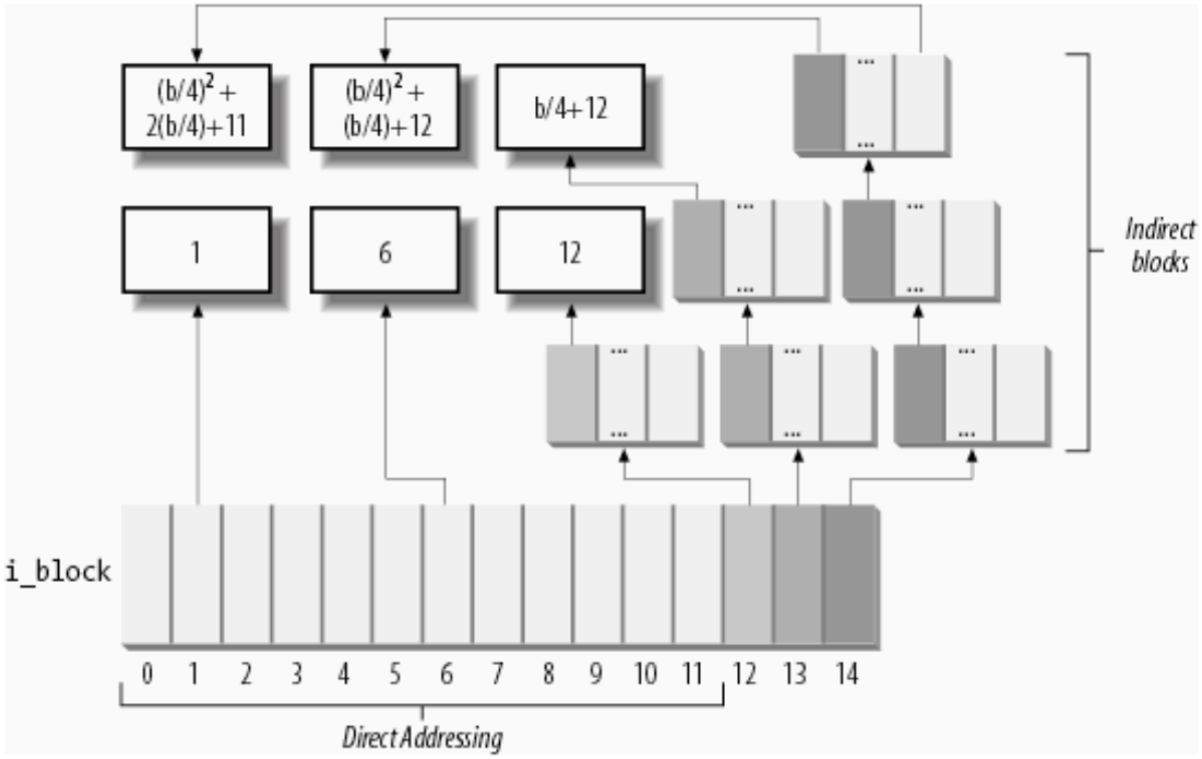
ext2

The general format is as shown:



Directory entries follow a linked list structure, making it slow to operate on large directories. This is addressed in ext3.

A file has pointers to the first 12 blocks, followed by an indirect block, a doubly indirect block and a trebly indirect block.



Ext2 file sizes are limited by its 32-bit **i_blocks** field. Their sizes can be 1KB, 2KB, 4KB, or 8KB.

Data Block Allocation

The system tries to allocate parent, children directories and their inner files to the same block group, allowing their accesses to be closely related.

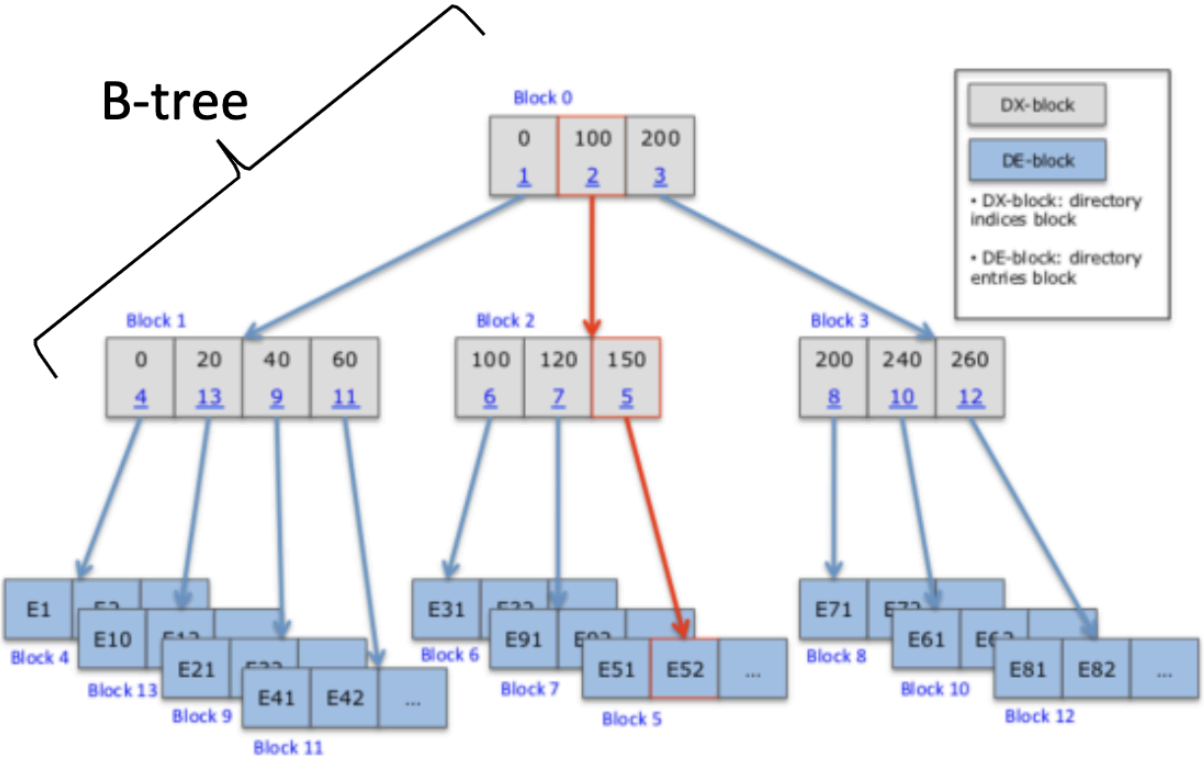
ext3

Mainly introduced **journaling** to help with system recovery. It is executed in 2 phases:

1. A copy of the blocks to be written is stored in the journal. When this is done, the blocks are written to the system.
2. When the blocks are finally written and I/O terminates, the copies of the blocks in the journal are **discarded**.

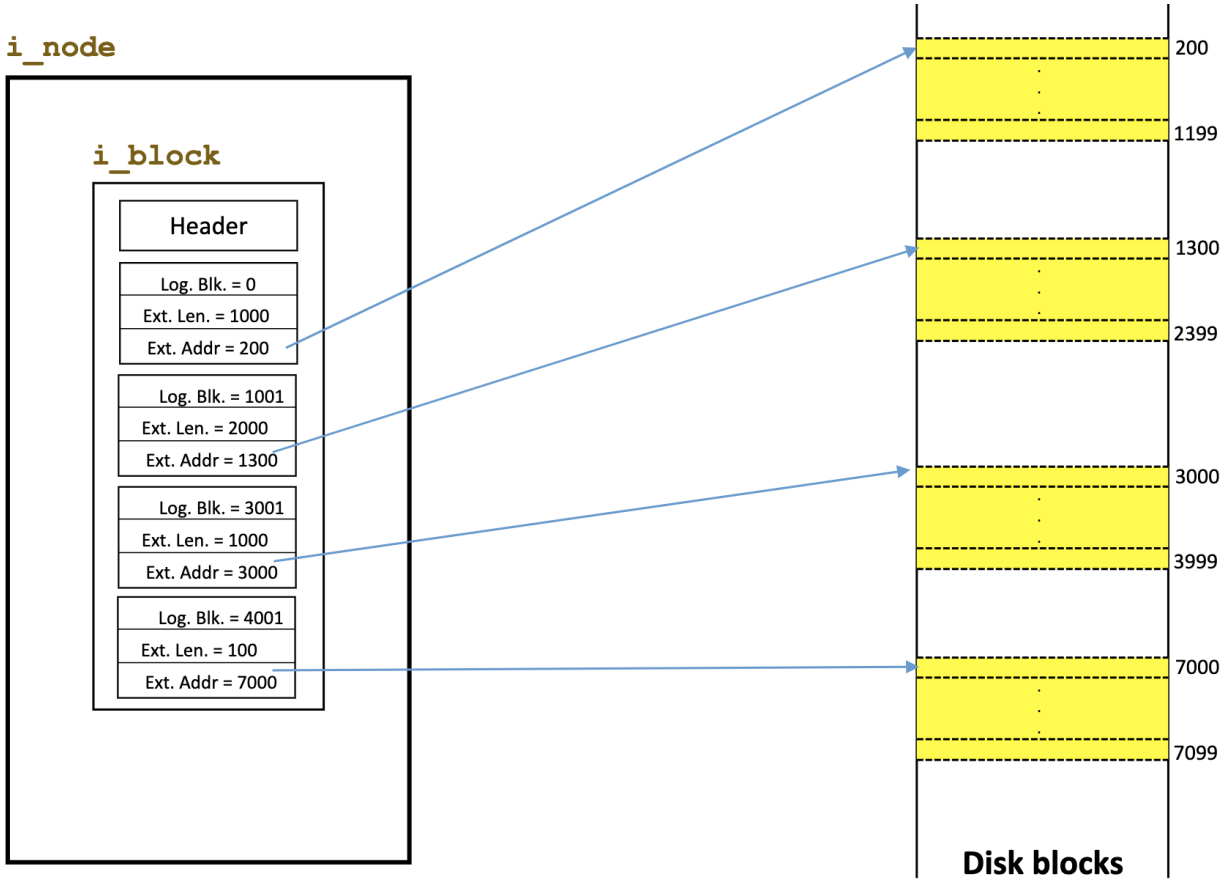
The information is stored in a hidden file **.journal** located at the root of the filesystem.

Directories use a **HTree**, which is also known as a hashed B-tree.



ext4

Introduced the **extent**. It is a **range of contiguous disk blocks**, which only stores the starting address and the length.

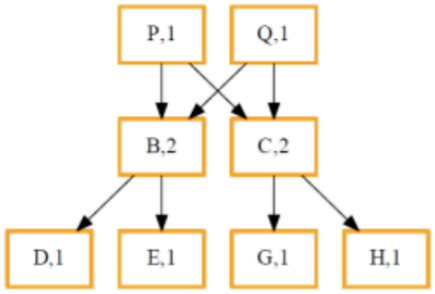


Other improvements of ex4:

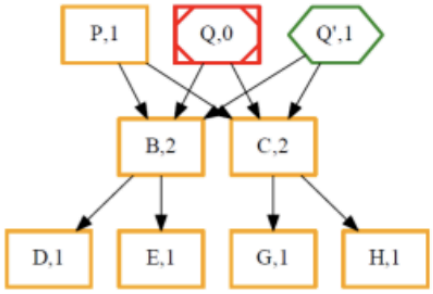
- Larger file sizes
- Pre-allocate disk space for a file
- Delaying block allocation until data flushed to disk
- Unlimited number of subdirectories
- Etc… Too many

B-tree Filesystem

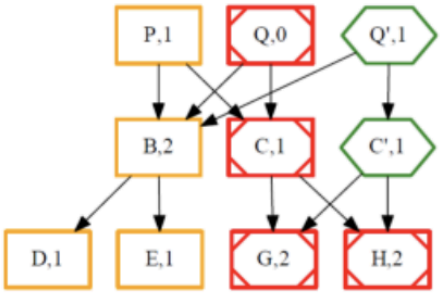
It uses a B-tree based filesystem, which is optimized for **copy-on-write** and **concurrency**. It is sorted on inode number.



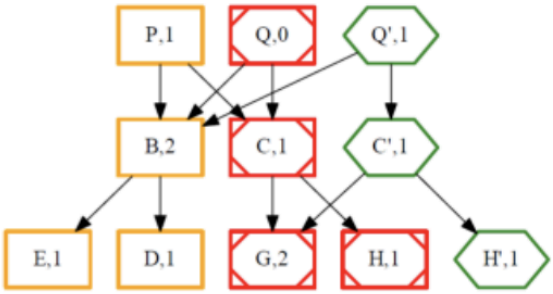
(a) Initial trees, T_p and T_q



(b) Shadow Q



(c) shadow C



(d) shadow H

Pointers are copied and a **reference count** is stored in each directory for **garbage collection**.

The B-tree only contains **keys, items and block headers**:

1. Block header stores metadata
2. Keys store object addresses
3. Items are keys with additional offset and size fields.

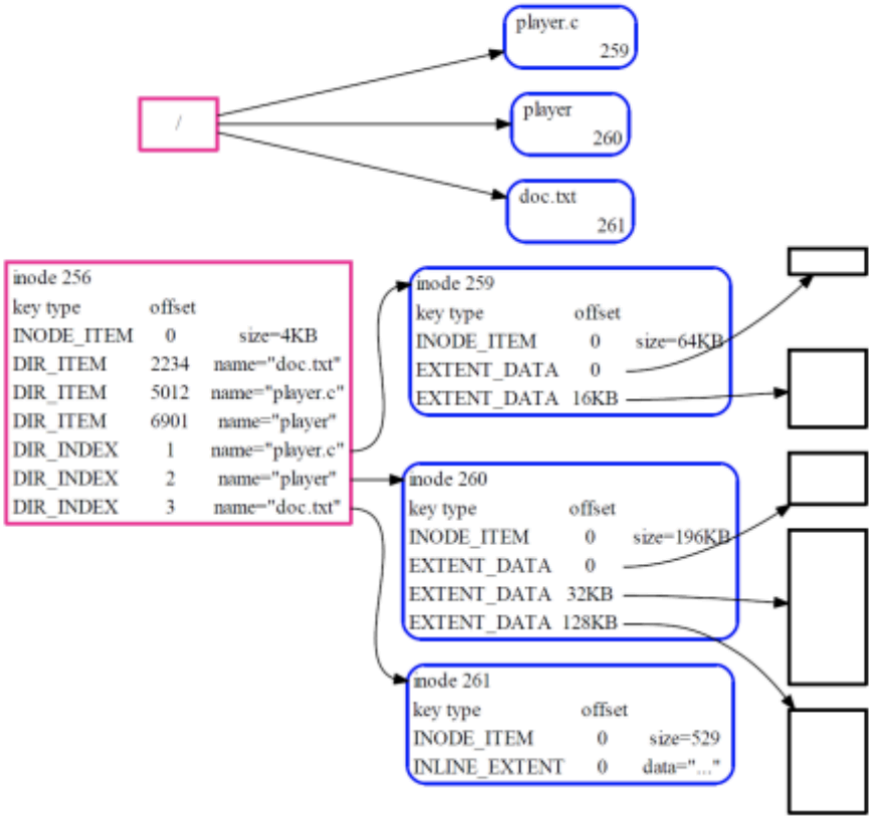
Internal nodes hold [key, block-pointer] pairs.
Leaf nodes hold arrays of [item, data] pairs.

Inode items are always the **first key**. They store metadata relating to the object.

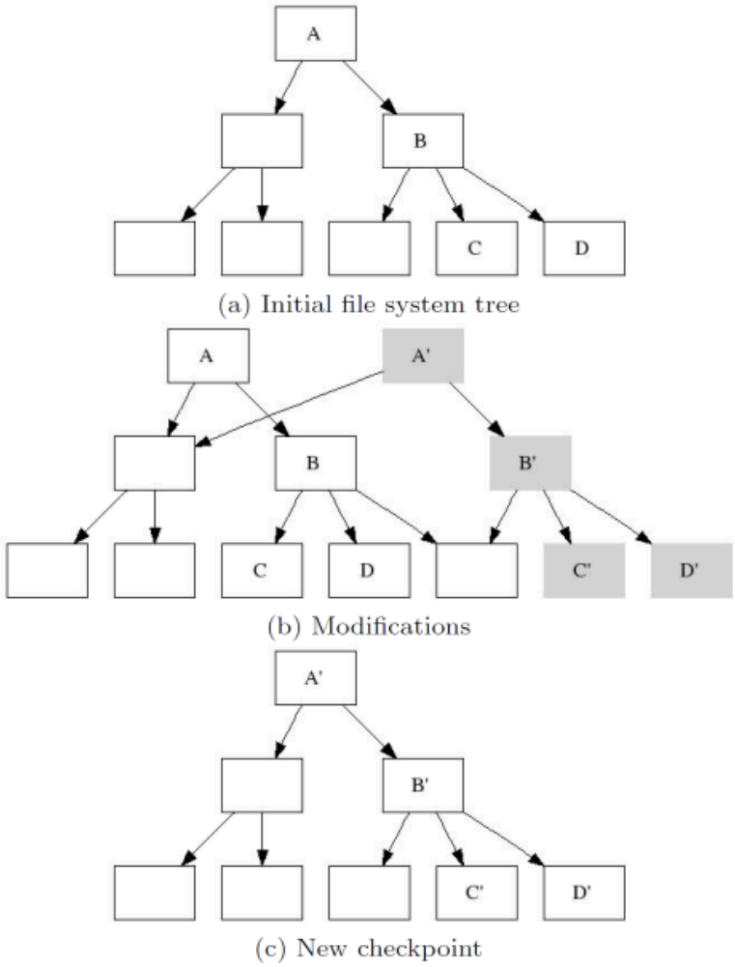
block header	I_0	I_1	I_2	free space	D_2	D_1	D_0

Figure 10: A leaf node with three items. The items are fixed size, but the data elements are variable sized.

Larger files are stored in **extent items**. Small files that occupy less than one leaf block may also be packed inside an extent.



Like databases, filesystems also have **checkpointing**.



Logical Volumes

A logical volume is a thin software layer on top of **hard disks and partitions**. They create an abstraction of continuity and ease-of-use by managing **hard drive replacement, repartitioning and backup**.

Device Drivers

Any subsystem output of processor and memory is considered I/O. Intel x86 uses a 16-bit **I/O port** to address I/O devices. Linux manages I/O ports in a tree-like structure rooted at **ioport_resource**.

4 special x86 instructions are used: **in, ins, out, outs**

They transfer data from a device to a register in the processor.

I/O Interface

Hardware circuitry inserted between a group of I/O ports and their corresponding device controller. May involve an IRQ line to the PIC to issue hardware interrupts.

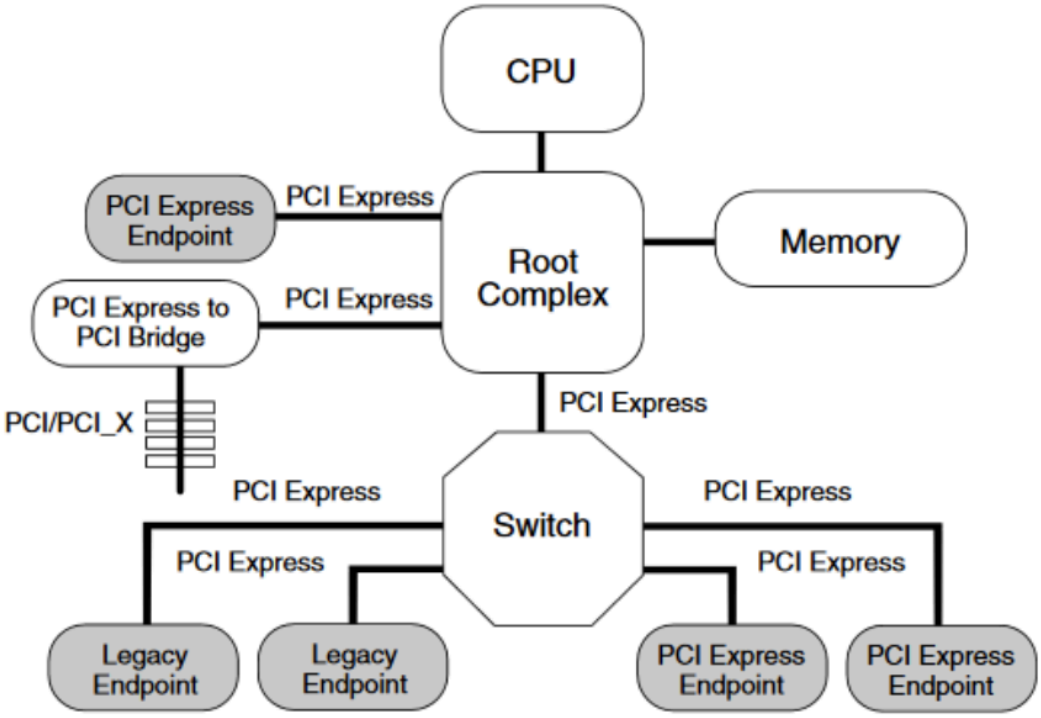
There are custom interfaces (keyboard, mouse, etc) and general purpose interfaces (USB, etc).

Device Controller

Converts commands from I/O interface to electrical signals from device (and vice versa to a status register in the CPU).

Memory-mapped I/O

I/O is embedded inside memory space, which is suitable for fast devices and DMA. An example would be the **PCI Express**, containing lanes with each lane having a full-duplex transport for a byte.

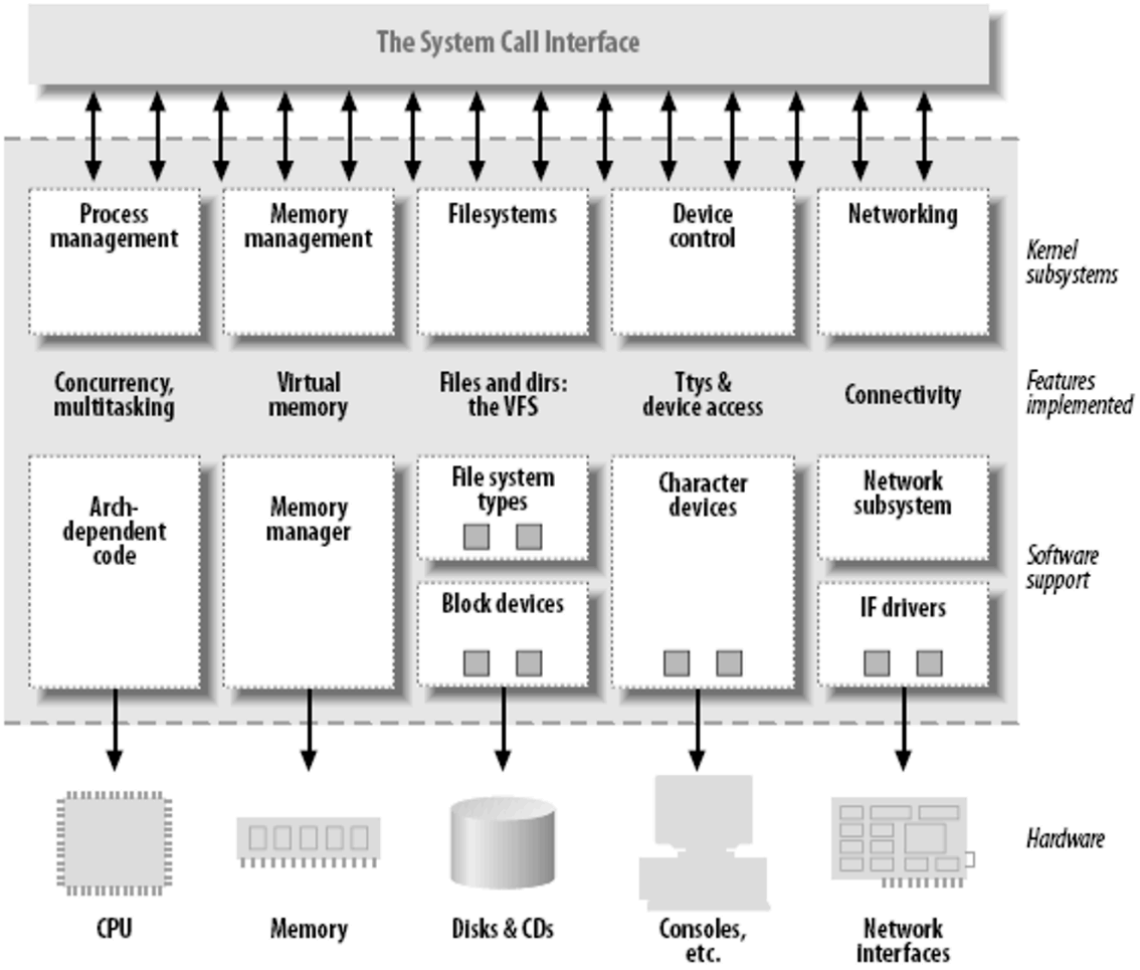


Role of Device Drivers

They implement **mechanisms** to access the hardware, but do not force particular **policies** on the user. For example, the driver cannot control how many people access the device, how it is accessed, etc. This makes it easier for the users and easier to write and maintain.

Drivers **map standard calls to device-specific operations** and can be plugged in at runtime as a **module** as needed.

The diagram below shows the role of I/O in the kernel.



Classes of Devices and Modules

The list is not exhaustive, there are other classes such as USB, SCSI, etc.

Character Devices

Returns a stream of bytes and can only be accessed sequentially (most of the time). Hence, they may not support file seeks.

They are usually simpler hardware with no caching, no re-reads of the same bytes. They can use buffering and the buffer is never reused, making it easier to manage.

Block Devices

Maps to an array of storage blocks and uses **buffering**. A block device can hold a filesystem. You can read more later on.

Network Devices

Helps to send and receive packets without knowing about individual connections.

File System Modules

They are **software drivers**, not device drivers. They are device-independent and serve as an abstraction for block devices.

Security issues

Problems can occur for:

- Setting up an interrupt line
- Setting up a default block size
- Buffer overrun
- Uninitialized memory
- Suspicious inputs or kernel

Version Numbering

Every software package in Linux has a release number.

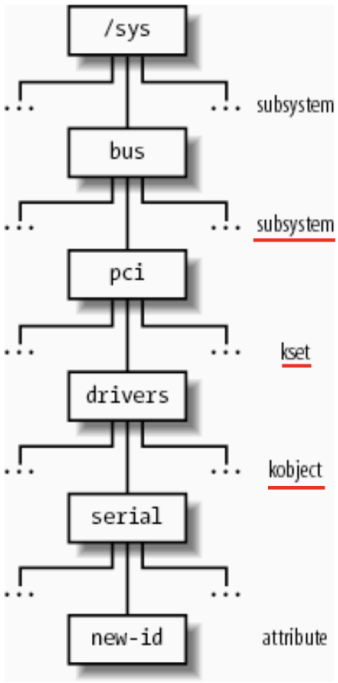
Before 3.0: **<major>.<minor>.<release>.<bugfix>**
After 3.0: **<version>.<release>.<bugfix>**

sysfs

Everything is treated as a file (except network cards).

Kobjects

They are the core data structure for device drivers. They are grouped into **ksets**. Each kobject corresponds to a directory in sysfs.



Components of the Linux device model

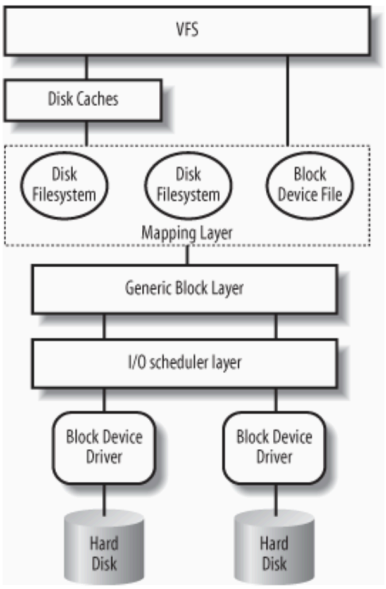
List of components:

1. **device** - for devices (associated with kobject)
2. **device_driver** - for drivers
3. **bus_type** - for buses
4. **class** - Represents a class of devices and belongs to **/sys/class**. A device can belong to multiple class devices. Drivers in the same class are expected to offer the **same functionalities** to the user mode applications.

Each logical device in the system should have an associated **device file** and a **well-defined device number**. The device number contains a 12-bit major number and a 20-bit minor number. It is allocated when a device driver registers, either statically or dynamically.

Block Devices

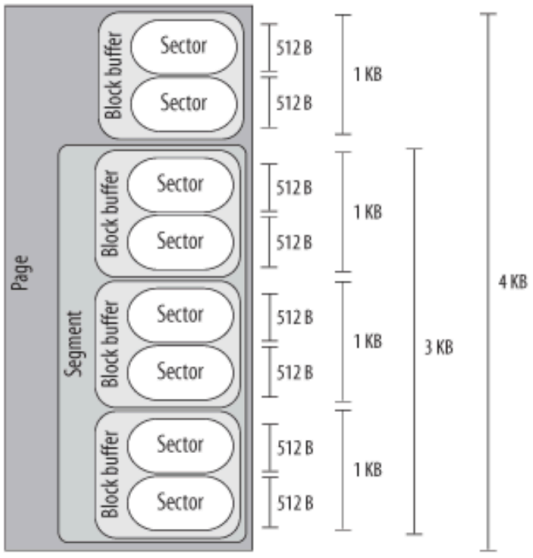
They repeatedly read/write the **same block of data**.



VFS needs to check if a block is already in some kind of cache. Otherwise, it needs to use the **mapping layer** to determine the physical location of the block. It will compute the **file block number** from the block size and use the inode data to determine the **logical block number** on disk.

Operations are issued to a **generic block layer** which issues **block I/O requests** to the **I/O scheduler layer**. Finally, the drivers can perform the actual operation on the device.

Size of data



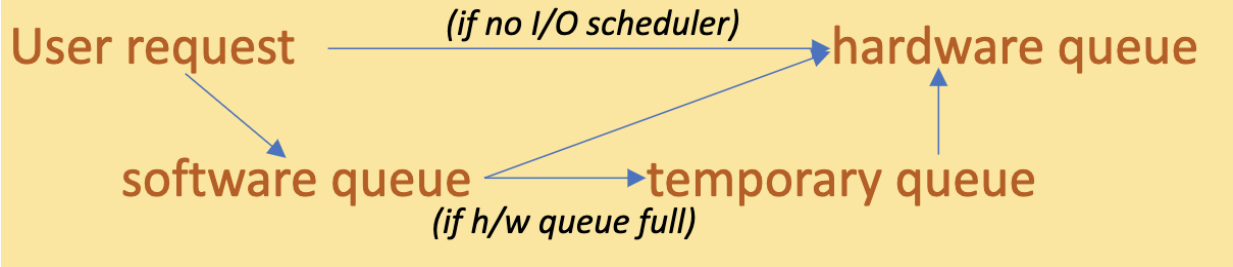
1. Hardware transfers data in **sectors**.
2. VFS uses logical units called **blocks**.
3. Block device drivers work with adjacent data in **segments**.
4. Disk caches work with **pages**.

Generic Block Layer

Data is placed into buffers. “Zero copy” is used to directly put data into user space.

Multi-Queue Block IO Queuing

Known as blk-mq. Has 2 versions for slow (hard disks) and fast (SSD, etc) devices respectively.



Hardware Dispatch Queues

This is the last step of block layer submission code, before the low level device driver takes over.

Software Staging Queues

This is for the I/O scheduler and ideally tries to **merge requests**. After that, the I/O scheduler will insert the request into a dispatch queue.

I/O Scheduler

Needs to maintain good performance while ensuring fairness of resource usage.

NOOP Elevator

It is a simple FIFO or LIFO.

Complete Fairness Queueing (CFQ) elevator

Maintains a default of 64 queues. Processes are **hashed** to queues by their PIDs and the queues are **sorted by sectors**.

Budget Fair Queueing I/O Scheduler (BFQ)

Guarantees responsiveness and low latency (prioritized over throughput).

Each process doing I/O on a device has a **weight** and a **queue**. Each queue has a **budget**.

Each process is given **exclusive access** to a device and issues requests until:

1. The queue is empty, with some extra time in case the process wants to issue more requests.
2. The budget is exhausted
3. Timeout

If a queue is marked as “low latency”, special heuristics will be used instead.

Early Queue Merge (EQM) is used to merge requests to a device. A request targeting a device is checked with the **next in line on the current active queue** (which may belong to another process). If hit, then merge.

Queues are scheduled based on **worst-case weighted fair queueing**, which is similar to CFS and also uses a red-black tree.

Multiqueue Deadline Elevator (mq-deadline)

Hinges on the idea of **deadlines**. There are 4 queues:

1. Read sorted by sector
2. Write sorted by sector
3. Read sorted by deadline
4. Write sorted by deadline

A request enters **both** its sector and deadline queue. A heuristic is first used to determine whether to read or write.

If the head of the deadline queue has expired, it should be dispatched. Adjacent requests are chosen from the sector queue.

If the request is not expired, continue on from the sector queue.

Read requests expire in 500 ms after entry into the queue while write requests expire in 5s.

Kyber I/O Scheduler

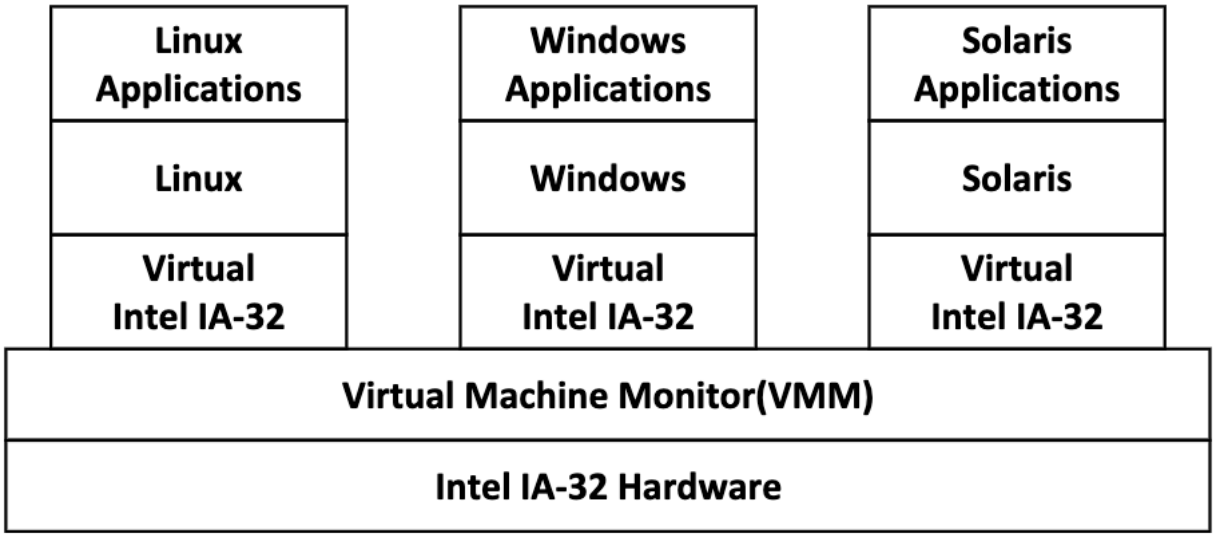
It is designed for **fast storage** like SSD, NVMe. It uses **tokens** to control the number of entries of each type in the hardware queue. By keeping the queue short, the amount of waiting time will be relatively small, ensuring a faster completion time for **higher-priority requests**. However, this comes at the cost of reducing opportunities for merging of requests, thus reducing throughput.

It has a **self-tuning** mechanism by measuring the completion time of each request and adjusting the time limits to achieve the desired latencies.

Virtualization

A (full) system virtualization can support multiple system images simultaneously, each running its own OS and application programs.

Real resources of the host platform are shared among the **guest system** with the virtual machine monitor (also known as the **hypervisor**).

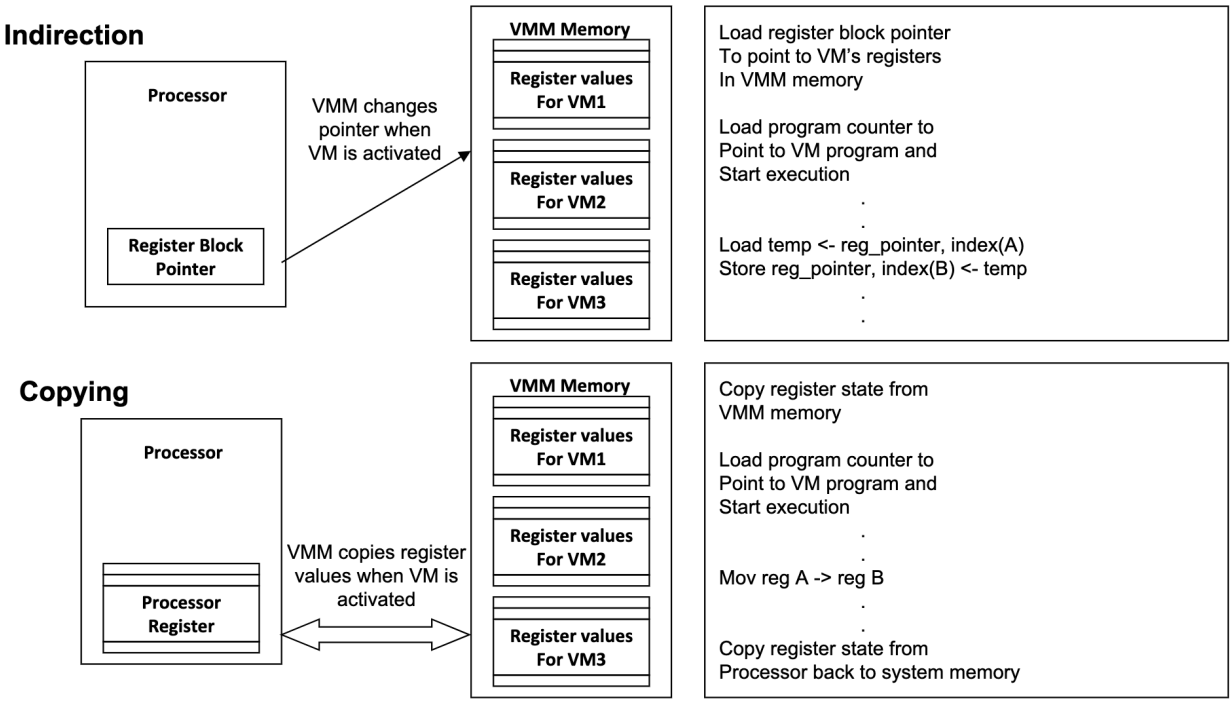


Other types of virtualizations:

1. HLL virtualization (eg. JVM)
2. Cross platform emulation (eg. QEMU, Appl Rosetta)
3. OS-level virtualization (eg. Docker, isolated user space instances)

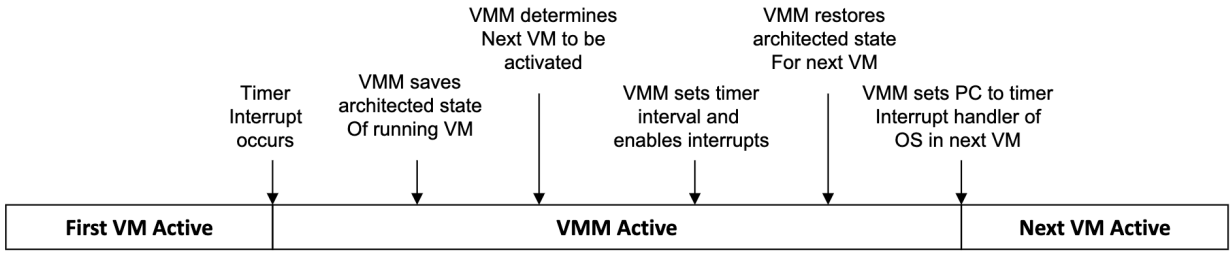
State Management

To give the illusion of dedicated hardware for a user in a VM, state management is required.

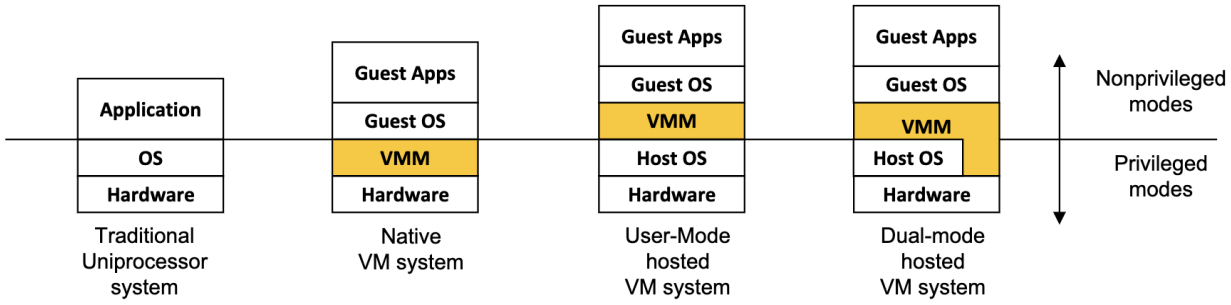


Resource Control

Time-sharing systems are used to balance usage of the physical hardware across VMs. Interrupts are eventually handled by VMM.



Types of VMs



Resource Virtualization

Processor

Need to virtualize the execution of guest instructions, including both system-level and user-level instructions. For **native systems**, VMM needs to run in system mode while other software runs in user mode. It will always use user mode in executing instructions from the guest VM.

Emulation

Uses interpretation and binary translation for different ISAs between guest and host.

Direct Native Execution

Used when there is identical ISA between guest and host, so as to achieve same performance when running on the VM.

Types of instructions

1. Control-sensitive: Attempt to change configuration of resources in the system.
2. Behavior-sensitive: Behavior or results produced depends on configuration of resource.
3. Innocuous: An instruction that is neither control or behavior sensitive.

Critical Instructions

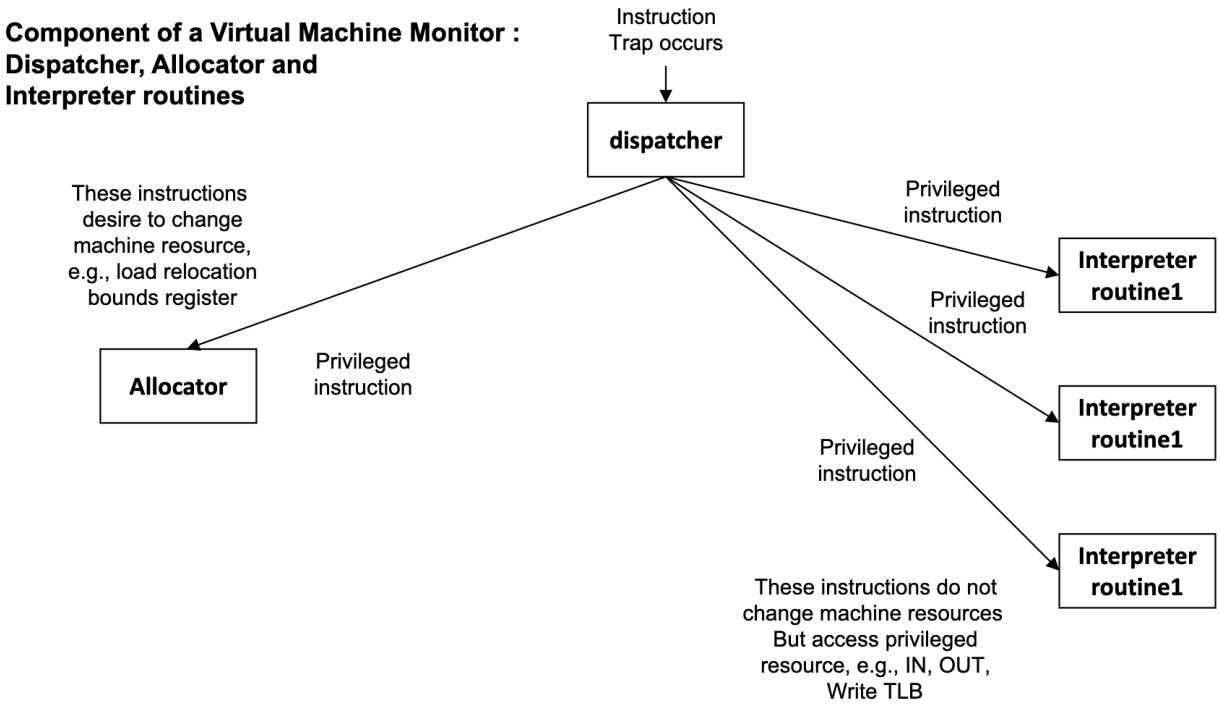
There are **problem instructions**. For example, **POPF** is **critical**, which is a sensitive instruction but not privileged, and thus **does not generate a trap in user mode**. Hence, additional steps must be taken by the VM to intercept the instruction (with some possible loss of efficiency).

IA-32 Architecture has 17 instructions that are critical. Hence, VM needs to be hybridized and requires **scanning and patching of critical instructions first**. This means that the VMM must scan the guest code and replace the critical instructions with a **trap or jump to the VMM**.

Critical instructions can be either **protection system references** or **sensitive register instructions**.

A cache is used to store the translation of emulation code.

At runtime, VMM will handle the instruction.



Keeping time

The **pvclock** protocol is used, which has a simple per-CPU structure that is shared between the host and guest. Basically, it allows the guests to get an accurate time.

I/O Devices

Dedicated Devices

Some devices can be **dedicated to a particular VM**, or switch from one guest to another after a very long time. Hence, the device does not necessarily have to be virtualized and requests can bypass the VMM.

Partitioned Devices

A very large disk can be partitioned for many VMs.

Shared Devices

Some devices, like a network adapter, can actually be shared by multiple VMs at a fine time granularity. Each guest may have to maintain its own virtual state (eg. virtual network address).

Nonexistent Physical Device

Some devices don't even need to exist. For example, you can emulate a network adapter used for communicating on the same host.

Virtualization at different levels for I/O

Basically, determines what level you intercept at.

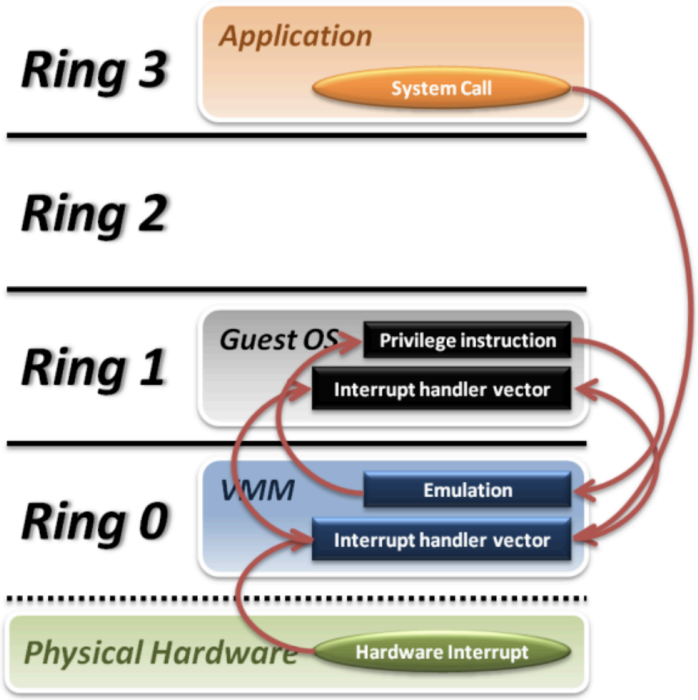
1. I/O Operation Level - VMM redirects I/O call to actual hardware.
2. Device Driver Level - VMM redirects the call to virtual driver to the physical driver.
3. System Call Level - VMM redirects the system call to the actual I/O.

Hardware support for virtualization

Trap and Emulate

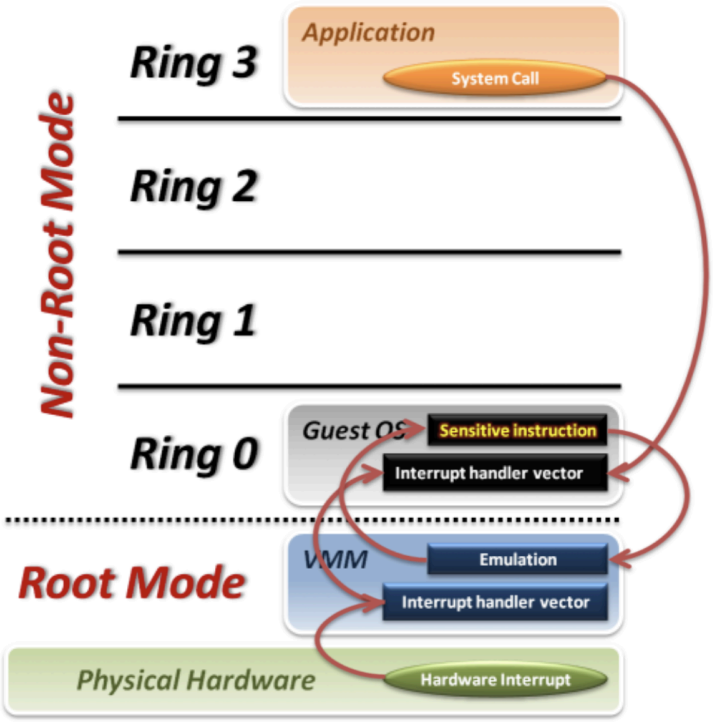
We cannot let guest OS handle everything, even though there might be no overhead at all if you did.

A solution would be **ring compression**. VMM is run in kernel mode, while kernel mode in the guest OS runs in ring 1. Now, VMM can intercept all trapping events.



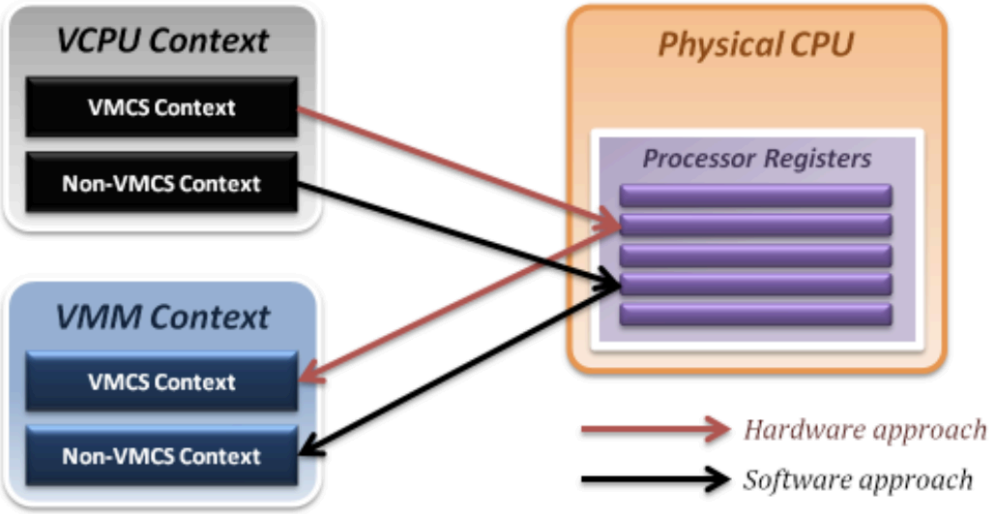
Intel VT-x

Adds one more **root mode** for the VMM. Now, sensitive instructions are redefined and will trap. There is also no need to trick the guest OS into thinking it's in Ring 0, because it is.



System State Management

The **VMCS** (Virtual Machine Control Structure) is used. It is similar to the TSS idea and is a more efficient hardware approach for register switching. When **VM Entry/Exit** occurs, the CPU automatically reads/writes the corresponding information into the VMCS.

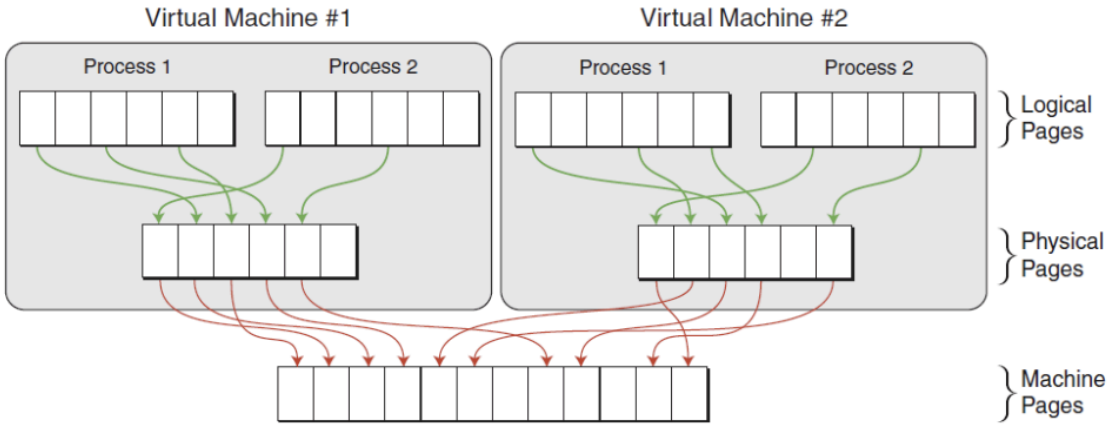


Memory

Extended Page Tables are used to maintain one set of page tables for each guest, also known as the **guest physical address**. Another set of page tables is maintained by the VMM, used to map the guest physical address to the **host physical address**.

For each memory access, the EPT MMU will:

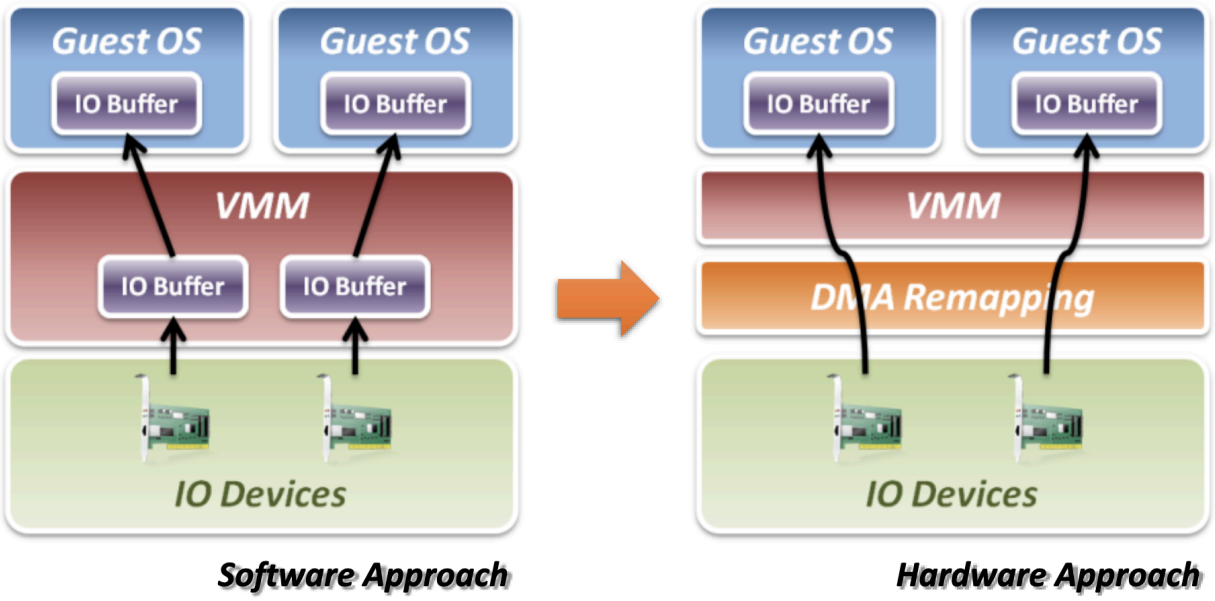
1. Get the guest physical address from the guest page table.
2. Get the host physical address by the VMM mapping table.



An **EPT violation** occurs when the VMM mapping table does not contain the entry for the guest physical address.

VT-d

It is a platform infrastructure for I/O virtualization and defines architecture for **DMA remapping**.



Linux KVM

Each VM has its own virtual BIOS, virtual hardware and exists as a Linux process in the host.