1. Segmentation and Paging

Young W. Lim

2021-07-05 Mon

Outline

1. Based on

2. Segmentation and Paging
   - Segments and Sections in ELF
   - Segmentation and Paging

Based on

"Study of ELF loading and relocs", 1999
http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

TOC: Segments and Sections in ELF

Sections (1)

- tell the linker if a section is either:
  - raw data to be loaded into memory,
    - e.g. `.data`, `.text`, etc., or
  - formatted metadata about other sections, used by the linker, but discarded at runtime
    - e.g. `.symtab`, `.srttab`, `.rela.text`

https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-s

Sections (2)

- size of each section except stack
  is specified in ELF file
- sections which are initialized from the ELF file
  - code (i.e., `.text`)
  - read-only data
  - initialized data segments
- other remaining sections are initially zero-filled
- sections have their own specified alignment

https://www.student.cs.uwaterloo.ca/~cs350/F07/notes/mem.pdf

Sections (3)

- sections comprise all information needed for linking a target object file in order to build a working executable.
- sections are needed on linktime but they are not needed on runtime.
- a Section Header Table : an array of `Elfxx_Shdr` structures, having one `Elfxx_Shdr` entry per section.

https://www.intezer.com/intezer-analyze/

Sections (4)

- Section Header Table Structure

sh_name	index of section name in section header string table
sh_type	section type
sh_flags	section attributes
sh_addr	virtual address of section
sh_offset	section offset in disk.
sh_size	section size.
sh_link	section link index.
sh_Info	Section extra information.
sh_addralign	section alignment.
sh_entsize	size of entries contained in section.

https://www.intezer.com/intezer-analyze/

Sections (5)

- some sections

`.text`	code
`.data`	initialised data
`.rodata`	initialised read-only data
`.bss`	uninitialized data
`.plt`	PLT (Procedure Linkage Table) (IAT equivalent)
`.got`	GOT entries dedicated to dynamically linked global variables
`.got.plt`	GOT entries dedicated to dynamically linked functions
`.symtab`	global symbol table
`.dynamic`	Holds all needed information for dynamic linking
`.dynsym`	symbol tables dedicated to dynamically linked symbols
`.strtab`	string table of .symtab section
`.dynstr`	string table of .dynsym section
`.interp`	RTLD embedded string
`.rel.dyn`	global variable relocation table
`.rel.plt`	function relocation table

`https://www.intezer.com/intezer-analyze/`

Segments (1)

- tells the operating system:
  - where should a segment be loaded into virtual memory
  - what permissions the segments have (`read`, `write`, `execute`). Remember that this can be efficiently enforced by the processor:

`https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-s`

Segments (2)

- a segment contains one or more sections
- the linker puts sections into segments
- a linker script (text file) can specify
  how sections are put into segments by ld in binutil

https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-s

Segments (3)

- <u>segments</u> are <u>page aligned</u>
- 3 segments
  - `.text, .rodata`
  - `.data. .bss, .sbss`
  - stack
- not all programs contain this many segments and sections

https://www.student.cs.uwaterloo.ca/~cs350/F07/notes/mem.pdf

Segments (4)

- Segments, which are commonly known as Program Headers, break down the structure of an ELF binary into suitable chunks to prepare the loading of the executable into memory.
- Program Headers are not needed on linktime.
- every ELF binary contains a Program Header Table which comprises of a single `Elfxx_Phdr` structure per existing segment.

https://www.intezer.com/intezer-analyze/

Segments (5)

- Program Header Table Structure

p_type	segment type.ELF Header
p_flags	segment attributes.
p_offset	file offset of segment.
p_vaddr	virtual address of segment.
p_paddr	physical address of segment.
p_filesz	size of segment on disk.
p_memsz	size of segment in memory.
P_align	segment alignment in memory.

https://www.intezer.com/intezer-analyze/

Segments (6)

- Some segment types

PT_NULL	unassigned segment usually first entry of Program Header Table
PT_LOAD	loadable segment
PT_INTERP	segment holding `.interp` section.
PT_TLS	Thread Local Storage segment common in statically linked binaries
PT_DYNAMIC	holding `.dynamic` section.

https://www.intezer.com/intezer-analyze/

Sections and segments (1)

- a section contain linktime information
  - static data for the linker
  - the section header table
- a segment contain runtime information
  - dynamic data for the OS
  - the program header (segment) table
- a segment can contain 0 or more sections

https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-s

Sections and segments (2)

- ELF files are composed of sections and segments
  - sections gather all needed information
    to link a given object file and build an executable
  - Program Headers split the executable
    into segments with different attributes,
    which will eventually be loaded into memory

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

Sections and segments (3)

- can consider segments as a tool to help the linux loader, as they group sections by attributes into single segments for the efficient loading process of the executable instead of loading each individual section into memory.

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

Sections and segments (4)

- offsets and virtual addresses of segments
  must be congruent modulo the page size
- their p_align field must be a multiple of the system page size.
- The reason for this alignment is to prevent
  the mapping of two different segments
  within a single memory page.

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

Sections and segments (5)

- different segments usually have different access attributes
- different segments cannot be mapped within the same memory page.
- the default segment alignment for PT_LOAD segments is usually a system page size
- The value of this alignment will vary in different architectures.

https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sectio

TOC: Segmentation and Paging

Structure of process address space

- text : program instructions
  - execute-only, fixed size
- data : variables (global, heap)
  - read/write, variable size
  - dynamic allocation by request
- stack : activation records
  - read/write, variable size
  - automatic growing / shrinking

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Segmented address space

- address space is a set of segments
- segment ; a linearly addressed memory
  - typically contains logically related information
  - program code, data, stack
- each segment has an identifier `s`, and a size `n`
  - $s \in [0, S-1]$, `S` = number of segments
- logical addresses are of form `(s, i)`
  - offset `i` within a segment `s`, and `i < n`

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Segmented address translation for segments

- segment table contains, for each segment s
  - base, bound, permission, valid bits
- logical address (s,i) to physical address translation
  - check if operation is permitted
  - check if `i < s.bound`
  - physical address = `s.base + i`

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Segmented address translation example

- 32-bit logical address
  - 10-bit segment `s`
  - 22-bit offset `i`
- segment table base register
- segment table bound register
- segment table entry
  - `v`, `perm`, `base`, `bound`
- `segtable[s].base + i`

`https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf`

Advantaes of segmentation

- each segment can be
    - located independently
    - separately protected
    - grow independently
- seqments can be shared between processes

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Problems of segmentation

- variable allocation
- difficult to find holes in physical memory
- must use one of non-trivial placement algorithms
  - first fit, next fit, best fit, worst fit
- external fragmentation

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Paged address space (1)

- address space is linear sequence of pages
  - page
  - physical unit of information
  - fixed size
- physicl memory is linear sequence of frames
  - a page fits exactly into a frame

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Paged address space (2)

- each page is identified by a page number 0 to N-1
  - N = number of pages in address space
  - N * page size = size of address space
- logical addresses are of form (p, i)
  - offset i within page p
  - i < page size

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Paged address translation for pages

- page table contains, for each page `p`
  - frame number that corresponds to `p`
  - perms, valid, reference, modified bits
- logical address `(p, i)` to physical address translation
  - check if operation is permitted
  - physical address = `p.frame + i`

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Paged address translation example

- 32-bit logical address
  - 22-bit page p
  - 10-bit offset i
- page table register
- page table entry
  - v, r, m, perm, frame #
- 32-bit physical address
  - pagep[p].frame + i

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Multi-level page tables

- 32-bit logical address
  - 12-bit page dir `d`
  - 10-bit page `p`
  - 10-bit offset `i`
- 32-bit physical address
  - `dir[d]->page[p].frame + i`

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Segmentation vs. paging

- segment is good logical unit of information
  - sharing, protection
- page is good physical unit of information
  - simple memory management
- combining both
  - segmentation on top of paging

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Cost of translation

- each page table costs a memory reference
  - for each reference, additional references required
  - slows machine down by factor of 2 or more
- take advantage of locality of reference
  - most references are to a small number of pages
  - keep translations of these in high speed memory
- problem
  - we don't know which pages until referenced

https://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf

Segmentation Unit (a)

- Segment Selector: It is the address present in Segment registers that will point to the particular Segment descriptor at a offset in GDT.
- Offset (Effective address): It is nothing but the memory address user see inside a program or anywhere in the system.
- Global Descriptor table: This is the table whose base address present in GDTR register and it contains Segment descriptors.
- Segment descriptor: It contains base phisical address(and few more info) from which offset is added to get the exact linear address.

`https://nixhacker.com/segmentation-in-intel-64-bit/`

Segmentation Unit (b)

- A segment selector is a 16 bit value held in a segment register. It is used to select an index for a segment descriptor from one of two tables.
  - GDT - Global Descriptor Table - for use system-wide
  - LDT - Local Descriptor Table - intended to be a table perprocess and switched when the kernel switches between process contexts
- There are six segment registers used to store these segment selector.
  - CS - Code Segment
  - SS - Stack Segment
  - DS - Data Segment
  - ES/FS/GS - Extra (usually data) segment registers

https://nixhacker.com/segmentation-in-intel-64-bit/

Segmentation Unit (c)

- For any kind of program execution to take place, at least CS, SS and DS segment registers must be loaded with valid segment selectors. Segment register also contains a hidden part along with segment selector that is used for caching purpose.

https://nixhacker.com/segmentation-in-intel-64-bit/

Segmentation Unit (d)

- Segment Selector Format
  15-3 : Index 2 : Table Indicator (0: GDT, 1:LDT) 1-0 : RPL (Requested Privilege Level)
- Table Indicator: Denotes if the descriptor that this particular selector points to is part of GDT table or LDT table.
- RPL: 2 bit. Can be between 0-3. It is the privilege level that the task (or segment selector of the task) has. We will talk more on this later.

https://nixhacker.com/segmentation-in-intel-64-bit/

Segmentation Unit (e)

- GDTR register holds a base address(32 bit in x32 and 64 bit in IA-32e(x64)) and 16-bits table limit for GDT. GDTR
- GDT is the table that contain all the Segment descriptors. Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level required for accessing the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segm

https://nixhacker.com/segmentation-in-intel-64-bit/

Segmentation Unit (f)

- A segment descriptor is mostly 8 bytes (or 16 byte for system segment in x64). The format looks like this:
- You can read more about each field in Intel developer's manual Vol-3a 3.4.5. We will cover only few important fields.
- Base (32 bits) - linear address where the segment starts
- Limit (20 bits) - Size of segment (either in bytes or 4kb blocks). End address of segment = base + limit.
- G (Granularity) flag - if 0, interpret limit as size in bytes. If 1, interpret as size in 4kb blocks.
- D/B - Default operation size flag. 0 = 16 bit default, 1 = 32 bit default. This is what actually controls whether an overloaded opcode is interpreted as dealing with 16 or 32 bit register/memory sizes
- DPL (Descriptor Privilege Level - 2 bits) - Specify the privilage level required by the descriptor. More on this in next section.

https://nixhacker.com/segmentation-in-intel-64-bit/

Segmentation Unit (g)

```
; offset 0x0
.null descriptor:
        dq 0

; offset 0x8
.code:            ; cs should point to this descriptor
dw 0xffff         ; segment limit first 0-15 bits
dw 0              ; base first 0-15 bits
db 0              ; base 16-23 bits
db 0x9a           ; access byte
db 11001111b      ; high 4 bits (flags) low 4 bits (limit 4 last bits)(limit is 20 bi
db 0              ; base 24-31 bits

; offset 0x10
.data:            ; ds, ss, es, fs, and gs should point to this descriptor
dw 0xffff         ; segment limit first 0-15 bits
dw 0              ; base first 0-15 bits
db 0              ; base 16-23 bits
db 0x92           ; access byte
db 11001111b      ; high 4 bits (flags) low 4 bits (limit 4 last bits)(limit is 20 bi
db 0              ; base 24-31 bits
```

https://en.wikipedia.org/wiki/Global_Descriptor_Table

Segmentation Unit (h)

- What "Limit 0:15" means is that the field contains bits 0-15 of the limit value.
- The base is a 32 bit value containing the linear address where the segment begins.
- The limit, a 20 bit value, tells the maximum addressable unit (either in 1 byte units, or in pages).
- Hence, if you choose page granularity (4 KiB) and set the limit value to 0xFFFFF the segment will span the full 4 GiB address space.

https://wiki.osdev.org/Global_Descriptor_Table

Segmentation Unit (i)

- In the Intel Architecture,
  and more precisely in protected mode,
  most of the memory management and
  Interrupt Service Routines are
  controlled through tables of descriptors
- Each descriptor stores information
  about a single object
  the CPU might need at some time.
  - a service routine
  - a task
  - a chunk of code or data

https://wiki.osdev.org/GDT_Tutorial

Segmentation Unit (j)

- for example, if you try to load a new value into a segment register, the CPU needs to perform safety and access control checks to see whether you're actually entitled to access that specific memory area.
- Once the checks are performed, useful values are cached in invisible registers of the CPU. (such as the lowest and highest addresses)

https://wiki.osdev.org/GDT_Tutorial

Segmentation Unit (k)

- Intel defined 3 types of tables:
  - the Interrupt Descriptor Table
    (which supplants the IVT)
  - the Global Descriptor Table (GDT)
  - the Local Descriptor Table
- each table is defined as a (size, linear address)
  to the CPU through the LIDT, LGDT, LLDT instructions, respectively.
- in most cases, the OS simply tells where those tables are
  once at boot time, and then simply goes writing/reading the tables
  through a pointer.

Segmentation Unit (I)

- LGDT / LIDT loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR).
- the source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the GDT or the IDT

https://www.felixcloutier.com/x86/lgdt:lidt

Segmentation Unit (m)

- If operand-size attribute is 32 bits,
  a 16-bit limit (lower 2 bytes of the 6-byte data operand) and
  a 32-bit base address (upper 4 bytes of the data operand)
  are loaded into the register.
- If the operand-size attribute is 16 bits,
  a 16-bit limit (lower 2 bytes) and
  a 24-bit base address (third, fourth, and fifth byte) are loaded.
- Here, the high-order byte of the operand is not used and
  the high-order byte of the base address in the GDTR or IDTR
  is filled with zeros.

https://www.felixcloutier.com/x86/lgdt:lidt

Segmentation Unit (n)

- The LGDT and LIDT instructions are used only in operating-system software;
- they are not used in application programs.
- They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode.
- They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

`https://www.felixcloutier.com/x86/lgdt:lidt`

Segmentation Unit (1)

- To translate a logical address into a corresponding linear address. the segmentation unit performs the following operations:
- Examines the `ti` field of the Segment Selector to determine which Descriptor Table (gdt / ldt) stores the Segment Descriptor

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

Segmentation Unit (2)

- `ti` field indicates that the Descriptor is
  - either in the GDT
    in this case, the segmentation unit gets
    the base linear address of the GDT from the gdtr register
  - or in the active LDT
    in this case the segmentation unit gets
    the base linear address of that LDT from the ldtr register

`https://www.halolinux.us/kernel-reference/segmentation-in-linux.html`

Segmentation Unit (3)

- Computes the address of the Segment Descriptor from the index field of the Segment Selector
- The index field is multiplied by 8 (the size of a Segment Descriptor), and the result is added to the content of the gdtr or ldtr register.
- Adds the offset of the logical address to the base field of the Segment Descriptor, thus obtaining the linear address.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

Segmentation Unit (4)

- Notice that, thanks to the nonprogrammable registers associated with the segmentation registers, the first two operations need to be performed only when a segmentation register has been changed.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

Segmentation Unit (5)

- a logical address
  - Selector [Index | TI]
  - Offset
- a descriptor location in gdt / ldt
  - base address
    - gdtr / ldtr <- TI in Selector
  - offset address
    - 8*Index in Selector
- a linear address
  - base address
    - descriptor content in gdt / ldt
  - offset address
    - offset of a logical address

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

GDT (General Descriptor Table)

- The 2.4 version of Linux uses segmentation
  only when required by the 80 x 86 architecture.
  In particular, all processes use the same logical addresses,
  so the total number of segments to be defined is quite limited,
  and it is possible to store all Segment Descriptors
  in the Global Descriptor Table (GDT).
  This table is implemented by the array gdt_table
  referred to by the gdt variable.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

LDT (Local Descriptor Table)

- Local Descriptor Tables are not used by the kernel, although a system call called modify_ldt( ) exists that allows processes to create their own LDTs. This turns out to be useful to applications (such as Wine) that execute segment-oriented Microsoft Windows applications.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A kernel code segments (1)

- The fields of the corresponding Segment Descriptor in the GDT
- `Base = 0x00000000`
- `Limit = 0xfffff`
- G (granularity flag) = 1, for segment size expressed in pages
- S (system flag) = 1, for normal code or data segment
- Type = 0xa, for code segment that can be read and executed
- dpl (Descriptor Privilege Level) = 0, for Kernel Mode
- D/B (32-bit address flag) = 1, for 32-bit offset addresses

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A kernel code segments (2)

- Thus, the linear addresses associated with that segment start at 0 and reach the addressing limit of 232 -1.
  The S and Type fields specify that the segment is a code segment that can be read and executed.
  Its dpl value is 0, so it can be accessed only in Kernel Mode.
  The corresponding Segment Selector is defined by the `__kernel_cs` `macro`.
  To address the segment, the kernel just loads the value yielded by the macro into the cs register.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A kernel data segments (1)

- The fields of the corresponding Segment Descriptor in the GDT
- Base = 0x00000000
- Limit = 0xfffff
  - G (granularity flag) = 1, for segment size expressed in pages
  - S (system flag) = 1, for normal code or data segment
  - Type = 2, for data segment that can be read and written
  - dpl (Descriptor Privilege Level) = 0, for Kernel Mode
  - D/B (32-bit address flag) = 1, for 32-bit offset addresses

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A kernel data segments (2)

- This segment is identical to the previous one (in fact, they overlap in the linear address space), except for the value of the Type field, which specifies that it is a data segment that can be read and written. The corresponding Segment Selector is defined by the `__kernel_ds` macro.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A user code segments (1)

- A user code segment shared by all processes in User Mode.
- The fields of the corresponding Segment Descriptor in the GDT
- Base = 0x00000000
- Limit = 0xfffff
  - G (granularity flag) = 1, for segment size expressed in pages
  - S (system flag) = 1, for normal code or data segment
  - Type = 0xa, for code segment that can be read and executed
  - dpl (Descriptor Privilege Level) = 3, for User Mode
  - D/B (32-bit address flag) = 1, for 32-bit offset addresses

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A user code segments (2)

- The S and dpl fields specify that the segment is not a system segment and its privilege level is equal to 3;
  it can thus be accessed both in Kernel Mode and in User Mode.
  The corresponding Segment Selector is defined
  by the `__USER_CS` macro.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A user data segments (1)

- A user data segment shared by all processes in User Mode.
- The fields of the corresponding Segment Descriptor in the GDT
- Base = 0x00000000
- Limit = 0xfffff
  - G (granularity flag) = 1, for segment size expressed in pages
  - S (system flag) = 1, for normal code or data segment
  - Type = 2, for data segment that can be read and written
  - dpl (Descriptor Privilege Level) = 3, for User Mode
  - D/B (32-bit address flag) = 1, for 32-bit offset addresses

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A user data segments (2)

- This segment overlaps the previous one:
  they are identical, except for the value of Type.
  The corresponding Segment Selector is defined
  by the `__user_ds` macro.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A default Local Descriptor Table (LDT) (1)

- A default Local Descriptor Table (LDT) that is usually shared by all processes.
  This segment is stored in the default_ldt variable.
  The default LDT includes a single entry consisting of a null Segment Descriptor.
  Each processor has its own LDT Segment Descriptor, which usually points to the common default LDT segment;
  its Base field is set to the address of default_ldt and its Limit field is set to 7.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

A default Local Descriptor Table (LDT) (2)

- When a process requiring a nonempty LDT is running, the LDT descriptor in the GDT corresponding to the executing CPU is replaced by the descriptor associated with the LDT that was built by the process.

https://www.halolinux.us/kernel-reference/segmentation-in-linux.html

Logical addresses in intel x86 (1)

- Whenever your program executes, CPU generates logical address for instructions which contains
- (16-bit segment selector, 32-bit offset)
- basically virtual (linear) address is generated using logical address fields

https://stackoverflow.com/questions/15851225/difference-between-physical-logical-v

Logical addresses in intel x86 (2)

- (16-bit segment selector, 32-bit offset)
- segment selector (identifier) refers to
  - code segment
  - data segment
  - stack segment etc.
- segment selector is 16-bit field
  - the first 13-bit is index
    a pointer to the segment descriptor resides in GDT
  - 1 bit TI field
    - TI = 1 Refer LDT (Local Descriptor Table)
    - TI = 0 Refer GDT (Global Descriptor Table)

https://stackoverflow.com/questions/15851225/difference-between-physical-logical-v

Logical addresses in intel x86 (3)

- Linux contains one GDT/LDT
  (Global/Local Descriptor Table)
  - contains 8 byte descriptor of each segments and
  - holds the base (virtual) address of the segment.
- So for for each logical address,
  virtual address is calculated using the following steps.

https://stackoverflow.com/questions/15851225/difference-between-physical-logical-v

Logical addresses in intel x86 (4)

1. examines the TI field of the segment selector
   to determine which descriptor table stores the segment descriptor
   TI field indicates that
   - the descriptor is in the GDT
     the segmentation unit gets the base linear address of the GDT
     from the gdtr register
   - the descriptor is in the active LDT
     the segmentation unit gets the base linear address of that LDT
     from the ldtr register

https://stackoverflow.com/questions/15851225/difference-between-physical-logical-v

2. Computes the address of the segment descriptor
   from the index field of the segment selector
   the index field is multiplied by 8 (the segment descriptor size),
   and the result is added to the content of the gdtr or ldtr register.
3. adds the offset of the logical address
   to the base field of the segment descriptor
   thus obtaining the linear (virtual) address.

Now it is the job of paging unit
to translate physical address from virtual address.

https://stackoverflow.com/questions/15851225/difference-between-physical-logical-v

Logical addresses in intel x86 (6)

- normally every address issued (for x86 architecture) is a logical address which is translated to a linear address via the segment tables.
- After the translation into linear address, it is then translated to physical address via page table.

https://stackoverflow.com/questions/15851225/difference-between-physical-logical-v