ELF1 7D Virtual Memory

Young W. Lim

2021-03-15 Mon

Outline

1. Based on
2. Virtual memory
   - Background
   - Segments and sections in ELF
   - Segmentation and Paging
   - Virtual memory
   - Kernal virtual / logical addresses
   - Kernel logical address
   - Kernel virtual address
   - User virtual address
   - Memory management unit
   - User space

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`

Logical addresses

- logical address
  - generated by CPU while a program is running
  - since it does not exist physically,
    it is also known as virtual address
  - used as a reference to access
    the physical memory location by CPU
- logical address space
  - the set of all logical addresses
    generated by a program's perspective.

https://www.geeksforgeeks.org/logical-and-physical-address-in-operating-system/

Physical addresses

- physical address
  - identifies a physical location in a memory
  - the user never directly uses the physical address
    but can access by the corresponding logical address.
- physical address space
  - all physical addresses corresponding
    to the logical addresses in a Logical address space

https://www.geeksforgeeks.org/logical-and-physical-address-in-operating-system/

Virtual addresses

- virtual addresses
  - the address you use in your programs,
  - the address that your CPU use to fetch data,
    is not real and gets translated via MMU
    to its corresponding physical address
- virtual address space
  - Linux running 32-bit has 4GB address space
  - each process has its own virtual address space

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

Memory managemet unit

- MMU (memory-management unit) hardware
  - maps logical address to its corresponding physical address
- OS along with MMU
  - the user program generates the logical address and
  - thinks that the program is running in this logical address
  - but to access physical memory for its execution, this logical address must be mapped to the physical address by MMU

https://www.geeksforgeeks.org/logical-and-physical-address-in-operating-system/

Logical vs virtual addresses (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 vs virtual addresses (2)

- 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-

Logical vs virtual addresses (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 vs virtual addresses (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 vs virtual addresses (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

Virtual address and physical address (1)

- Physical addresses are provided directly by the machine
  - one physical address space per machine
  - addresses typically range
    from some minumum (sometimes 0) to some maximum,
  - some portions of this range are usually used
    by the OS and/or devices,
    and not available for user processes

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

Virtual address and physical address (2)

- Virtual addresses (or logical addresses) are addresses provided by the OS
  - one virtual address space per process
  - addresses typically start at zero, but not necessarily
  - space may consist of several segments

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

Virtual address and physical address (3)

- address translation (or address binding) means mapping virtual addresses to physical addresses

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

High / Low memory

- Low memory
  Memory for which logical addresses exist in kernel space.
  On almost every system you will likely encounter,
  all memory is low memory.

- High memory
  Memory for which logical addresses do not exist,
  because it is beyond the address range
  set aside for kernel virtual addresses.

https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch15.html

kmap (1)

- kmap returns a kernel virtual address
  for any page in the system.
  - for low-memory pages
    it just returns the logical address of the page;
  - for high-memory pages,
    creates a special mapping in a dedicated part
    of the kernel address space.

https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch15.html

kmap (2)

- Mappings created with kmap should always be freed with kunmap;
- a limited number of such mappings is available,
  so it is better not to hold on to them for too long.
- kmap calls maintain a counter,
  so if two or more functions both call kmap on the same page,
  the right thing happens.
- Note also that kmap can sleep if no mappings are available.

https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch15.html

Sections and sections (1)

- 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 and sections (2)

- segments are page aligned
- 3 segments = (`.text` + `.rodata`), (`.data` + `.sbss` + `.bss`), (stack)
- not all programs contain this many segments and sections

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

Sections and sections (3)

- the segments contain information needed at runtime, while the sections contain information needed during linking.
- A segment can contain 0 or more sections
- Section contains static for the linker, segment dynamic data for the OS

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

Sections and sections (4)

- to understand the fields of the section header and program header (segment) entries, and how they are be used by the linker (sections) and operating system (segment).

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

Sections and sections (5)

- section: 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,
    that will be used by the linker, but disappear at runtime
    e.g. .symtab, .srttab, .rela.text

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

Sections and sections (6)

- segment: 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: How does x86 paging work?

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

Sections and sections (7)

- a segment contains one or more sections
- it is the linker that puts sections into segments.
- in binutils, how sections are put into segments by ld is determined by a text file called a linker script.

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

Sections (1)

- 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 (2)

- 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 (3)

- 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)

- 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.
- In contrast with Section Headers, Program Headers are not needed on linktime.
- On the other hand, similarly to Section Headers, 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 (2)

- 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 (3)

- 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)

- 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 (2)

- 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 (3)

- 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 (4)

- this is due to the fact that
  different segments usually have different access attributes,
- this is not possible if two segments are 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

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

TLB (translation lookaside buffer) (1)

- Virtual Memory would not be very effective
  if every virtual memory address had to be translated
  by looking up the associated physical page in memory.
- the solution is to cache the recent translations
  in a Translation Lookaside Buffer (TLB)

https://courses.cs.washington.edu/courses/cse378/00au/Lec28.pdf

TLB (translation lookaside buffer) (2)

- the TLB is a small cache of
  the most recent virtual-physical mappings
- by checking here first, temporal locality is exploited
  to speed virtual address transaltion
  - while a virtual-to-physical translation is under way,
    the hardware checks to see if it has seen
    this translation recently

https://courses.cs.washington.edu/courses/cse378/00au/Lec28.pdf

TLB (translation lookaside buffer) (3)

- fast associative memory
  keeps most recent translations
  (logical page, page frame)
- determine whether non-offset part of LA
  (logical address) is in TLB (translation lookaside buffer)
  - if so, get corresponding frame num for physical address
  - if not, wait for normal memory translation (parallel)

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

Translation cost with TLB

- cost is determined by
  - speed of memory : ~ 100 nsec
  - speed of TLB : ~ 20 nsec
  - hit ratio : fraction of refs satisfied by TLB, ~95%
- Speed with no address translation : 100 nsec
- Speed with address translation
  - TLB miss : 200 nsec (100% slowdown)
  - TLB hit : 120 nsec (20% slowdown)
  - avarage : $120 * .95 + 200 * .05 = 124$ nsec

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

TLB design issues

- the larger the TLB
  - the higher the hit ratio
  - the slower the response
  - the greater the expense
- TLB has a major effect on performance
  - must be flushed on context switches
  - alternative : tagging entries with PIDs
- MIPS: has only a TLB, no page tables
  - devote more chip space to TLB

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

MMU and TLB

- Memory Management Unit (MMU)
  - hardware unit that translates a virtual address to a physical address
  - every memory reference is passed through the MMU
- Translation Lookaside Buffer (TLB)
  - a cache for the *MMU*'s virtual-to-physical translations table
  - not needed for correctness
  - but source of significant performance gain

https://cseweb.ucsd.edu/classes/su09/cse120/lectures/Lecture7.pdf

Single address space (1)

- simple systems
- sharing the same memory space
  - memory and peripherals
  - all processes and OS
- no memory proctection

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Single address space (2)

- CPUs with single address space
  - 8086 - 80286
  - ARM Cortex-M
  - 8 / 16-bit PIC
  - AVR
  - most 8- and 16-bit systems

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Single address space (3)

- portable c programs expect flat memory
  - multiple memory access methods limit portability
- management is tricky
  - need to know / detect total RAM
  - need to keep processes separated
- no protection

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual memory (1)

- a system that uses an address mapping
- maps virtual address space to physical address space
  - to physical RAM
  - to hardware devices
    - PCI devices
    - GPU RAM
    - On-SOC IP blocks

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual memory (2)

- Advantages
  - each process can have a different memory mapping
    one process' RAM is invisible to other processes
    built in memory protection
    kernel RAM is invisiable to user space processes
  - memory can be moved
  - memory can be swapped to disk

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

- Advantages (continued)
  - hardware device memory can be mapped
    into process' address space
    requires the kernel to perform the mapping
  - physical RAM can be mapped
    into multiple processes at once
    shared memory
  - memory regions can have access permissions
    read / write / execute

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual memory (4)

- Physical addresses
  addresses used by the hardware (DMA, peripherals)
- Virtual addresses
  addresses used by software
  - RISC: load/store instructions
  - CISC: any instruction accessing memory

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual memory (5)

- mapping is performed in hardware
- no performance penalty
  for accessing already mapped RAM regions
- permissions are handled without penalty
- the same instructions are used
  to access RAM and mapped hardware
- software will only use virtual addreses
  in its normal operation

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

MMU (Memory Management Unit) (1)

- MMU is the hardware responsible for implementing virtual memory
- sits between the CPU core and memory
- usually the part of the physical CPU on ARM, it's part of the licensed core
- separate from the RAM controller DDR controller is a separate IP block

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

MMU (Memory Management Unit) (2)

- transparently handels all memory accesses
  from load / store instructions
- maps memory acceses using virtual addresses
  to system RAM and peripheral hardware
- handles permissions
- generates an exception (page fault)
  on an invalid access

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

TLB (Translation Lookaside Buffer)

- TLB is consulted by the MMU
  when CPU accesses a virtual address
- if the virtual address is in the TLB,
  the MMU can look up the physical address
- if the virtual address is not in the TLB,
  the MMU will generate a page fault exception
  and interrupt the CPU
- if the virtual address is in the TLB,
  but the permissions are insufficient,
  the MMU will generate a page fault

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Page faults

- a page fault is a CPU exception generated when software attempts to use an invalide virtual address
  - the virtual address is not mapped for the process requesting it
  - the processes has insufficient permissions for the address
  - the virtual address is valide, but swapped out

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel virtual address (1)

- in linux, the kernel uses virtual addresses as user space processes do
  this is not true of all OS's

- virtual address space is split
  1. the upper part is used for the kernel
  2. the lower part is used for user space
  3. 32-bit linux have the split address 0xc0000000

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel virtual address (2)

- By default, the kernel uses the top 1GB of virtual address space
- each user space process gets the lower 3GB of virtual address space

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual addresses - linux (1)

- kernel address space is the area above CONFIG_PAGE_OFFSET
- for 32-bit, this is configurable at kernel build time
  - the kernel can be given a different amount of address space as desired
- for 64-bit, the split varies by architecture but it is high enough

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual addresses - linux (2)

- three kinds of virtual addresses in Linux
- Kernel
  - Kernel Logical Address
  - Kernel Virtual Address
- User Space
  - User Virtual Address

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel virtual / logical addresses (1)

- the kernel maps most of the *kernel virtual address space* to perform 1:1 mapping with an offset of the top part of physical memory (3GB - 4GB)
  - slightly *less then* for 1Gb for 32bit x86
  - can be different for other processors or configurations
- for kernel code on x86 address 0xc00000001 is mapped to physical address 0x1.
- This is called logical mapping
  - a 1:1 mapping (with an offset) that allows the kernel to access most of the physical memory of the machine.

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel virtual / logical addresses (2)

- in the following cases, the kernel keeps a region at the top of its virtual address space where it maps a "random" page
  - when we have *more then* 1Gb physical memory on a 32bit machine,
  - when we want to reference non-contiguous physical memory blocks as contiguous
  - when we want to map memory mapped IO regions
- this mapping does not follow the 1:1 pattern of the logical mapping area.
- This is called the virtual mapping.

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel virtual / logical addresses (3)

- on many platforms (x86 is an example),
  both the logical and virtual mapping are done
  using the same hardware mechanism
  (TLB controlling virtual memory).
- In many cases, the logical mapping is actually done
  using virtual memory facility of the processor,
  (this can be a little confusing)
- The difference is in which mapping scheme is used:
  - 1:1 for logical
  - random for virtual (paging)

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel virtual / logical addresses (4)

- 3 kinds of addressing

1. Logical Addressing : Address is formed by base and offset
   This is nothing but segmented addressing,
   where the address (or offset) in the program is always used
   with the base value in the segment descriptor

1. Linear Addressing : also called virtual address
   Here virtual adresses are contigous,
   but the physical address are not contiguous
   Paging is used to implement this.

1. Physical Addressing : the actual address on the Main Memory

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel virtual / logical addresses (5)

- in linux, the kernel memory (in address space) is beyond 3 GB, i.e. 0xc000000.
- the addresses used by Kernel are not physical addresses
  - to map the virtual address from 3GB to 4GB it uses `PAGE_OFFSET`.
    - no page translation is involved.
    - contiguous address
    - except 896 MB on x86.
  - beyond the address space from 3GB to 4GB, paging is used for translation.
    - `vmalloc` returns these addresses

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel virtual / logical addresses (6)

- when virtual memory is referred
  in context of user space,
  then it is through paging
- if kernel memory is mentioned
  then it is the address mapped
  - by `PAGE_OFFSET` (kernel logical address)
  - by `vmalloc` (kernel vitual address)

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel virtual / logical addresses (7)

- PAGE_OFFSET in x86 is 0XC0000000,
  (1100_0000_0000_0000_0000_0000_0000_0000)
  or 3 gigabytes ($3 * 2^{30}$)
- this is where the 3G/1G split is defined.
- every address above PAGE_OFFSET
  is the kernel virtual address
- any address below PAGE_OFFSET
  is a user space address

https://linux-mm.org/VirtualMemory

Kernel virtual / logical addresses (8)

- to get kernel memory in byte-sized chunks.
  - `kmalloc()`
    virtually contiguous
    physically contiguous
  - `vmalloc()`
    virtually contiguous
    not necessarily physically contiguous

https://stackoverflow.com/questions/116343/what-is-the-difference-between-vmalloc-

Kernel virtual / logical addresses (9)

- On a 32-bit system, `kmalloc()`
  - returns the kernel logical address (it is a virtual address)
  - the direct mapping (constant offset)
  - a contiguous physical chunk of RAM.
  - suitable for DMA where we give only
- `vmalloc()`
  - returns the kernel virtual address
  - paging (not direct mapping)
  - not necessarily a contiguous chunk of RAM
  - Useful for large memory allocation and
    in cases where non-contiguous physicl memory is allowed

https://stackoverflow.com/questions/116343/what-is-the-difference-between-vmalloc-

Kernel virtual / logical addresses (10)

- kernel logical addresses use
  normal CPU memory access functions.
- On 32-bit systems,
  only 4GB of kernel logical address space exists,
  even if more physical memory than that is in use.
- logical address space supported by physical memory
  can be allocated with `kmalloc()`

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel virtual / logical addresses (11)

- kernel virtual addresses do not necessarily have corresponding logical addresses.
- You can allocate physical memory with `vmalloc` and get back a virtual address that has no corresponding logical address (on 32-bit systems with PAE, for example).
- use `kmap()` to assign a logical address to that virtual address.

https://stackoverflow.com/questions/8708463/difference-between-kernel-virtual-addr

Kernel logical addresses (1)

- normal address space of the kernel
  `kmalloc()`
- virtual addresses are a fixed offset
  from their physical addresses
  virtual 0xc0000000 → physical 0x00000000
- easy conversion between physical and virtual addresses

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel logical addresses (2)

- kernel logical addreses can be converted to and from physical addresses using these macros
  `__pa(x)`
  `__va(x)`
- for small memory systems (less than 1G of RAM) kernel logical address space starts at `PAGE_OFFSET` and goes through the end of physical memory

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel logical addresses (3)

- kernel logical address space includes
  - memory allocated with `kmalloc()` and most other allocation methods
  - kernel stacks per process
- kernel logical memory can never be swapped out

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel logical addresses (4)

- kernel logical addresses use a fixed mapping between physical and virtual address space
- this means virtually contiguous regions are by nature also physically contiguous
- this combined with inability to be swapped out, makes them suitable for DMA transfers

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel logical addresses (5)

- for 32-bit large memory systems (> 1GB RAM)
  not all of the physical RAM can be mapped
  into the kernel's address space
- kernel address space is the top 1GB of
  virtual address space, by default
- upto 104 MB is reserved at the top of
  the kernel memory space
  for non-contiguous allocation
  `vmalloc()`

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel logical addresses (6)

- in a large memory case,
  only the bottom part of physical RAM
  is mapped directly into
  kernel logical address space
- only the bottom part of physical RAM has
  a kernel logical address
- this case is never applied to 64-bit systems
  - there is always enough kernel address space
    to accommodate all the RAM

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Low and High Memory

- low memory
  - physical memory which has a kernel logical address
  - physically contiguous
- high memory
  - physical memory beyond -~896MB
  - has no logical address
  - not physically contiguous when used in the kernel
  - only on 32-bit

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel virtual addresses (1)

- kernel virtual addresses are
  above the kernel logical address mapping
- kernel virtual addresses - `vmalloc()`
- kernel logical addresses - `kmalloc()`

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel virtual addresses (2)

- kernel virtual addresses are used for
  - non-contiguous memory mappings
    - often for large buffers which could potentially be too large to find contiguous memory
    - `vmalloc()`
  - memory-mapped I/O
    - map peripheral devices into kernel
    - PCI, SoC IP blocks
    - `ioremap()`, `kmap()`

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Kernel virtual addresses (3)

- the important difference is that memory in the kernel virtual address area (`vmalloc()` area) is non-contiguous physically
- this makes it easier to allocate, especially for large buffers on small memory systems
- this makes it unsuitable for DMA

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

- in a large memory situation,
  the kernel virtual address area is smaller,
  because there is more physical memory
- an interesting case, where more memory means
  less space for kernel virtual addresses
- in 64-bit, of course, this doesn't happen,
  as PAGE_OFFSET is large, and
  there is much more virtual address space

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

User virtual addresses (1)

- represent memory used by user space programs
  - the most of the memory on most systems
  - where the most of the compilation is
- all addresses below `PAGE_OFFSET`
- each process has its own mapping
  - threads share a mapping
  - complex behavior with `clone(2)`

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

User virtual addresses (2)

- kernel logical addresses use a fixed mapping
  user space processes make full use of the MMU
  - only the used portions of RAM are mapped
  - memory is not contiguous
  - memory may be swapped out
  - memory can be moved

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

User virtual addresses (3)

- since user virtual addresses are not guaranteed to be swapped in, or even allocated at all,
- user buffers are not suitable for use by the kernel (or for DMA), by default
- each process has its own memory map `struct mm` pointers in `task_struct`
- at context switch time, the memory map is changed this is part of the overhead

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Memory management unit (1)

- the MMU manages virtual address mappings
  - maps virtual addresses to physical addresses
- the MMU operates on basic units of memory : pages
  - page size varies by architecture
  - some architectures have configurable page sizes

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Memory management unit (2)

- common page sizes
  - ARM - 4k
  - ARM64 - 4k or 64k
  - MIPS - widely configurable
  - x86 - 4k

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Memory management unit (3)

- a page is
  - a unit of memory size
  - aligned at the page size
  - abstract
- a page frame refers to
  - a physical memory block
    which is page sized and page aligned
  - physical
- the pfn (page frame number) is often used to refer to physical page frames in the kernel

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Memory management unit (4)

- the MMU operates on pages
- the MMU maps physical frames to virtual addresses
- a memory map for a process contains many mappings
- a mapping often covers multiple pages
- the TLB holds each mapping
  - virtual address
  - physical address
  - permissions

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Page faults

- when a process acceses a region of memorythat is not mapped, the MMU will generate a page fault exception
- the kernel handles page fault exceptions regularly as part of its memory management design
- TLB can contain only the part of the required maps for a process
- page faults at context switch time
- lazy allocation

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Basic TLB mappings (1)

- user virtual address space
  - mapped pages unmapped space
- physical address space
  - allocated frames
- TLB mapings
  - TLB entries (page, page frame)
  - virtually contiguous regions
    not physically contiguous

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Basic TLB mappings (2)

- mappings to virtually contiguous regions do not have to be physically contiguous
- easy memory allocation
- almost all user space code does not need physically contiguous memory

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Multiple processes

- each *process* has its own set of mappings
- the same virtual addresses in two different processes will likely be used to map different physical addresses
  - (page, page frame1) for process 1
  - (page, page frame2) for process 2

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Shared memory (1)

- shared memory is easily implemented with an MMU
- simply map the same physical frame
  into two different processes
- the virtual addresses need not be the same
  - for pointers to values inside a shared memory region
    the virtual addresses must be the same

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Shared memory (2)

- the shared memory region can be mapped to different virtual addresses in each process
- the `mmap()` system call allows the user space process to request a specific virtual address to map the shared memory region
  - if the kernel cannot grant a mapping at this address, `mmap()` returns with failure

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Lazy allocation (1)

- the kernel does not allocate pages immeidately that are requested by a process
- the kernel will wait until those pages are actually used
- lazy allocation to optimize a performance
  - if the requested pages may not be actually used, then the allocation will never happen

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Lazy allocation (2)

- when memory is requested for allocation,
  the kernel simply creates
  a record of the *request* in its page tables
  and then returns (quickly) to the process,
  without updating the TLB
- when that newly-allocated memory is actually accessed,
  the CPU will generate a page fault,
  because the CPU doesn't know about the mapping
  (no entry in the TLB)

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Lazy allocation (3)

- in the page fault handler,
  the kernel uses its page tables
  to determine that the mapping is valid
  (from the kernel's point of view)
  yet unmapped in the TLB
- the kernel will allocate a physical page frame
  and update the TLB with the new mapping
- the kernel returns from the exception handler and
  user space program can resume

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Lazy allocation (4)

- in a lazy allocation case, the user space program is never aware that the page fault happened
- the page fault can only be detected at the time that was lost to handle it
- for processses that are time-sensitive pages can be pre-faulted, or simply touched, at the start of execution
  - see also `mlock()` and `mlockall()`

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Page tables (1)

- the entries in the TLB are a limited resource
- far more mappings can be made than can exist in the TLB at one time
- the kernel must keep track of all of the mappings at all times
- the krenel stores all these informations in the page tables `stuct_mm` and `vm_area_struct`

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Page tables (2)

- since the TLB can only hold a limited subset of the total mappings for a process, some valid mappings will not have TLB entries
- when these addresses are touched the CPU will generate a page fault because the CPU has no knowledge of the mapping only the kernel does

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Page tables (3)

- the page fault handler will
  - find the appropriate mapping for the offending addresses in the krenel's page tables
  - select and remove an existing TLB entry
  - create a TLB entry for the page containing the address
  - return to the user space process
    - observe the similarities to lazy allocation handling

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Swapping (1)

- when memory utilization is high,
  the kernel may swap some frames to disk
  to free up RAM
- the MMU makes this possible
  - the kernel may copy a frame to disk and remove its TLB entry
  - the frame may be reused by another process

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Swapping (2)

- when the frame is needed again,
  the CPU will generate a page fault
  because the address is not in the TLB

- at a page fault time, the kernel can
  - put the process to sleep
  - copy the frame from the disk
    into an unused frame in RAM
  - fix the page table entry
  - wake the process

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Swapping (3)

- note that when the page is restored to RAM,
  it is not necessarily restored to the same physical frame
  where it originally was located (before being swapped out)
- the MMU will use the same virtual address though,
  so the user space program will not know the difference
    - this is why user space memory cannot
      typically be used for DMA

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

User space

- there are several ways to allocate memory from user space
  - ignoring the familiar `*alloc()` functions, which sit on top of platform methods
- `mmap()` can be used directly to allocate and map pages
- `brk()` / `sbrk()` can be used to increase the heap size

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

mmap()

- mmap() is the standard way to allocate large amounts of memory from user space
- while mmap() is often used for files, the MAP_ANONYMOUS flag causes mmap() to allocate normal memory for the process
- the MAP_SHARED flag can make the allocated pages sharable with other processes

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

brk() / sbrk() (1)

- `brk()` sets the top of the program break
- this is the top of the data segment
  but inspecton of `kernel/sys.c` shows
  it separates from the data segment
- this in effect increases the size of the heap
- `sbrk()` increases the program break
  rather than setting it directly

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

brk() / sbrk() (2)

- lazy allocation
- see mm/mmap.c for do_brk()
- do_brk() is implemented similar to ~mmap()
- modify the page tables for the new area
- wait for the page fault
- optionally, do_brk() can pre-fault the new area
  and allocate it
  see mlock(2) to control this behavior

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

- malloc() and calloc() will use either brk() or mmap() depending on the requested allocation size
    - small allocations use brk()
    - large allocaion use mmap()
    - see mallopt(3) and the M_MMAP_THRESHOD parameter to control this behavio

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Stack

- Stack expansion
- if a process accesses memory beyond its stack,
  the CPU will trigger a page fault
- the page fault handler detects
  the address is just beyond the stack, and
  allocates a new page to extend the stack
- the new page will not be physically contiguous
  with the rest of the stack
- see `__do_page_fault()` in `/arch/arm/mm/fault.c`

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf