ELF2 1D Virtual Memory

Young W. Lim

2021-12-27 Mon

Outline

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: Virtual Memory in Operating System

- Single address space
- Virtual memory
- Demand paging
- Swapping
- Page fault

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) address mapping

- virtual memory
  a system that uses an address mapping
- maps virtual address space to physical address space
  - to physical memory 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) memory management techqnique

- virtual memory is a memory management technique
  - prvoides an idealized abstraction
    of the storage resources
    that are actually available on a given machine
  - creates the illusion to users
    of a very large (main) memory
  - main storage, as seen by a process or task,
    appears as a contiguous address space or
    collection of contiguous segments.

https://en.wikipedia.org/wiki/Virtual_memory#Paged_virtual_memory

Virtual memory (3) virtual address sapce

- the OS using a combination of HW and SW
  - maps a program's memory addresses (virtual addresses), into physical addresses in computer memory.
- the OS manages
  - virtual address spaces and
  - the assignment of real memory to virtual memory
- a virtual address space
  - may exceed the capacity of real memory
  - can reference greater memory than physical memory

https://en.wikipedia.org/wiki/Virtual_memory#Paged_virtual_memory

Virtual memory (4) memory management unit

- mapping is performed in hardware
- Memory Managemet Unit
  - address translation hardware in the CPU
  - automatically translates
    virtual addresses to physical addresses

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

Virtual memory (5) memory management unit

- software will only use virtual addreses
  - in its normal operation
- the same instructions
  - for accessing RAM and mapped hardware
- no performance penalty
  - in accessing already mapped RAM regions
  - in handling permissions

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

Virtual memory (6) process and address space

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

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

Virtual memory (7) paging / segmentation

- paging or segmentation techniques enable
  - to use more memory than physically available
  - to share memory used by libraries between processes
  - to free applications from managing a shared memory space
  - to increase security due to memory isolation

https://en.wikipedia.org/wiki/Virtual_memory#Paged_virtual_memory

Virtual memory (8) memory protection

- each process can have its own 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

Demand Paging (1)

1. If CPU try to refer a page
   that is currently not available in the main memory
   it generates an interrupt indicating memory access fault
2. The OS puts the interrupted process in a blocking state
   to continue the execution, the OS must bring
   the required page into the memory
3. The OS will search for the required page
   in the logical address space

https://www.geeksforgeeks.org/virtual-memory-in-operating-system/

Demand Paging (2)

4. The required page will be brought from logical address space to physical address space.
    - the page replacement algorithms are used for the decision making of replacing the page in physical address space.
5. The page table will updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into ready state

https://www.geeksforgeeks.org/virtual-memory-in-operating-system/

Swapping

- swapping a process out means
  removing all of its pages from memory
  - removed by the normal page replacement process.
- suspending a process ensures that
  it is not runnable while it is swapped out.
- At some later time, the system swaps back the process
  from the secondary storage to main memory
- when a process is busy swapping pages in and out
  then this situation is called thrashing

https://www.geeksforgeeks.org/virtual-memory-in-operating-system/

Page faults

- a page fault is a CPU exception
  generated when software attempts
  to access an invalid 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 valid, but swapped out

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

TOC: Address Types

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/

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/`

Virtual addresses

- virtual addresses
  - the address you use in your programs
    - the address that your CPU use to fetch data is not real (virtual)
    - must be 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

Physical address space - one per machine

- 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 *parts* 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 space - one per process

- 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*
- address space may consist of several segments

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

Logical vs. virtual addresses (1) assembly program

- logical addresses are those seen and used by the assembly programmer
- physical addresses are those directly corresponding to the logic levels of the hardware address bus
  - logical and physical addresses can be indentical in simple systems
  - the MMU converts logical addresses into physical addresses
  - the conversion scheme vary with the architecture of the system.

https://electronics.stackexchange.com/questions/178681/difference-between-logical-

Logical vs. virtual addresses (2) conversion

- logical and physical addresses can be indentical in simple systems
- the MMU converts logical addresses into physical addresses
- the conversion scheme vary with the architecture of the system.

https://electronics.stackexchange.com/questions/178681/difference-between-logical-

Logical vs. virtual addresses (3) conversion scheme

- widely used conversion schemes are
  those employing virtual memory
  - paging and segmentation
  - the logical address in such a system is
    also called virtual address.
- a virtual address is a logical address
  on a system with virtual memory
  - virtual memory is called this way
    because a logical (i.e. virtual) address does
    not necessarily map to an actual physical address

https://electronics.stackexchange.com/questions/178681/difference-between-logical-

- the memory content addressed by a virtual address
  - could reside only on disk
  - have to be brought into main memory
    before it can be used
- on a system without virtual memory
  a logical address always maps to a physical address,
  - to some "real" memory e.g. RAM, ROM
  - to some register in a device
    if memory-mapped I/O is implemented

https://electronics.stackexchange.com/questions/178681/difference-between-logical-

Logical vs. virtual addresses (5) demand paging

- consider an access to a virtual address only a tentative access
  - if the memory content being accessed is already in physical memory, access is granted
  - Otherwise an interrupt is generated (page miss) and an interrupt routine is called
    - will load the needed memory page into physical memory from its actual location (typically the page file on disk).
    - is also responsible for freeing some physical memory if there is no room left to load the requested page.

https://electronics.stackexchange.com/questions/178681/difference-between-logical-

Logical vs. virtual addresses (6) CPU vs RAM addresses

- Logical address
  the address as the CPU instructions are using.
  there can be many more such addresses than
  there is RAM (or other memory or IO) in the system.
- Physical address
  the address that is sent to the RAM (or ROM, or IO)
  for a read or write operation.

https://electronics.stackexchange.com/questions/178681/difference-between-logical-

Logical vs. virtual addresses (7) translate or interrupt

- For a simple system, physical address = logical address
- larger systems are generally demand-paged virtual memory systems,
  - the MMU translates a logical address to a physical address,
  - or alerts the OS (_interrupt_) to take action
    - to allocate a page,
    - read a page from disk, or
    - deny access to a page -> trap or fault

https://electronics.stackexchange.com/questions/178681/difference-between-logical-

TOC: GNU ELF Addresses

ELF object files

- the linker combines *input* files
  into a single *output* file
  - input object files
  - output object / executable file
  - all in object file format

`https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/`

ELF sections

- each object file has a list of sections
  - input sections
  - output sections
- each section in an object file has
  - a *name*
  - a *size*
  - section contents :
    most sections are associated with block of data

https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/B

ELF section types

- a loadable section
  - the contents should be *loaded* into memory when the output file is *run*
- an allocatable section
  - a section with no contents may be *allocatable*
  - an area in memory should be *set aside* but nothing should be *loaded* there
  - in some cases, this memory must be filled with /zero/es
- sections for debugging

https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/I

ELF sections vs segments (1)

- section: tell the linker if a section is either:
  - raw data to be loaded into memory,
    - e.g. `.data`, `.text`, etc.
  - 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

ELF sections vs segments (2)

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

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

ELF sections (1)

- As we have mentioned, sections make up segments. Sections are a way to organise the binary into logical areas to communicate information between the compiler and the linker. In some special binaries, such as the Linux kernel, sections are used in more specific ways
- We've seen how segments ultimately come down to a blob of data in a file on disk with some descriptions about where it should be loaded and what permissions it has. Sections have a similar header to segments

https://www.bottomupcs.com/elf.xhtml

ELF sections (2)

Section Header

```
typedef struct {
        Elf32_Word   sh_name;
        Elf32_Word   sh_type;
        Elf32_Word   sh_flags;
        Elf32_Addr   sh_addr;
        Elf32_Off    sh_offset;
        Elf32_Word   sh_size;
        Elf32_Word   sh_link;
        Elf32_Word   sh_info;
        Elf32_Word   sh_addralign;
        Elf32_Word   sh_entsize;
} ELF32_Shdr;
```

https://www.bottomupcs.com/elf.xhtml

ELF sections (3)

- Sections have a few more types defined for the sh_type field; for example a section of type SH_PROGBITS is defined as a section that hold binary data for use by the program. Other flags say if this section is a symbol table (used by the linker or debugger for example) or maybe something for the dynamic loader. There are also more attributes, such as the allocate attribute which flags that this section will need memory allocated for it.

`https://www.bottomupcs.com/elf.xhtml`

Load and run addresses (1)

- The load address is the location of an object
  in the load image
- The run address is the location of the object
  as it exists during program execution
- An object is a chunk of memory.
  It represents a section, segment, function, or data.

https://downloads.ti.com/docs/esd/SPRU513/load-and-run-addresses-slau1317366.html

Load and run addresses (2)

- The load and run addresses for an object may be the same
  - This is commonly the case for program code and read-only data, such as the `.econst` section.
    - the program can *read the data* directly from the load address
  - sections that have no initial value, such as the `.ebss` section
    - do not have load data
    - considered to have the same load and run addresses
    - if you specify different load and run addresses for an uninitialized section, the linker provides a warning and ignores the load address.

https://downloads.ti.com/docs/esd/SPRU513/load-and-run-addresses-slau1317366.html

Load and run addresses (3)

- The load and run addresses for an object may be different.
  - This is commonly the case for writable data, such as the `.data` section.
  - The `.data` section's starting contents are placed in ROM and *copied to* RAM.
  - This often occurs during program startup, but depending on the needs of the object, it may be deferred to sometime later in the program

https://downloads.ti.com/docs/esd/SPRU513/load-and-run-addresses-slau1317366.html

p_vaddr and p_paddr (1)

- p_vaddr is a virtual address,
  p_paddr is a physical address.
- these are the addresses at which
  the *data in the file* will be loaded.
- they map the contents of the file
  into their corresponding memory locations

https://stackoverflow.com/questions/16812574/elf-files-what-is-a-section-and-why-d

p_vaddr and p_paddr (2)

- physical addresses are the raw memory addresses.
  - on modern operating systems,
    physical addresses are no longer used in the user space
    Instead, user space programs use virtual addresses.
- the OS deceives that
  - the user space program uses the memory alone,
  - the entire address space is available for it.
- the OS maps those virtual addresses
  to physical addresses in the actual memory,
  and it does it transparently to the program.

https://stackoverflow.com/questions/16812574/elf-files-what-is-a-section-and-why-d

p_vaddr and p_paddr (3)

- not every address in the virtual address space is available simultaneously
- limited by the actual physical memory available.
- the OS just maps the memory for the segments the program actually uses
- if the process tries to access some unmapped memory, the operating system incurs memory access fault (The program can address it, but it cannot access it)

https://stackoverflow.com/questions/16812574/elf-files-what-is-a-section-and-why-d

LMA & VMA (1)

- every loadable or allocatable output section has two addresses.
- the VMA (Virtual Memory Address)
  - the address the *output section* will have when the output file is run
- the LMA (Load Memory Address)
  - the address at which the *output section* will be loaded

https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/I

LMA & VMA (2)

- in most cases, VMA and LMA will be the same
- VMA and LMA might be different
  when a *data section* is loaded from ROM,
  and then copied into RAM when the program starts up
  - this technique is often used to initialize global variables
    in a ROM based system
  - in this case the ROM address would be the LMA
    and the RAM address would be the VMA

`https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/B`

LMA & VMA (3)

- The section header contains a single address.
- the address in the section header is the VMA
- The program headers contain the mapping of VMA to LMA
- `objdump -x`

https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect

LMA & VMA (4)

ELF section header example

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
<a few lines removed>
  3 .bss          00000004  00000048  0000018c  00000240  2**1
                  ALLOC
```

- .bss has a VMA 0x048
- .bss has a LMA 0x18c

https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect

LMA & VMA (5)

ELF program header example

```
Program Header:
<a few lines removed>
    LOAD off    0x00000240 vaddr 0x00000048 paddr 0x0000018c align 2**0
         filesz 0x00000000 memsz 0x00000004 flags rw-
```

- a vaddr of 0x048 (VMA)
- a paddr of 0x18c (LMA)

https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect

LMA & VMA (6)

- ELF file segment does have the physical address attribute
  ELF file section does not have physical address attribute.
- It is possible though to map sections
  to corresponding segment memory.
- The meaning of physical address is architecture dependent
  and may vary between different OS's and hardware platforms.

https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect

LMA & VMA (7)

- VMA and LMA are GNU utility terminology
  not in the ELF specification.
- an ELF executable file has program header fields :
  - p_paddr
  - p_vaddr

https://stackoverflow.com/questions/39888381/elf-loading-when-vma-lma

LMA & VMA (8)

p_vaddr

- this member gives the virtual address
  at which the first byte of the segment resides in memory

p_paddr

- on systems for which physical addressing is relevant,
  this member is reserved for the segment's physical address
- because System V ignores physical addressing
  for application programs,
  this member has unspecified contents
  for executable files and shared objects.

https://refspecs.linuxbase.org/elf/gabi4+/ch5.pheader.html

LMA & VMA (9)

- by default, ARM IDE DS-5 uses p_vaddr, which is the standard
- Usage of p_paddr is a quality of implementation, and is left very loosely defined in the specification.
- The ARM Compiler, Linker and C Library does not generate this information (p_vaddr, p_paddr) since the relocation process is handled internally (scatter loading).

https://stackoverflow.com/questions/39888381/elf-loading-when-vma-lma

LMA & VMA (10)

- some environments use p_paddr
  - not as a physical address,
  - but the load address (hence LMA),
- some use p_paddr
  - as an address to resolve symbols
    before and after MMU is enabled

https://stackoverflow.com/questions/39888381/elf-loading-when-vma-lma

TOC: Kernel Addresses

Kernel address in linux (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 address in linux (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

Kernel address in linux (3)

- 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

Kernel address in linux (4)

- 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 address in linux (5)

- Consider a 32bit x86 Linux system with 4 GB of RAM memory
  - Kernel logical address
    - upto 896 MB
    - allocated using `kmalloc()`
    - one to one mapped
  - Kernel virtual address
    - 128MB (1024-896, above 896MB kernel logical address)
    - allocated using `vmalloc()`
    - virtually contiguous but physically non-contiguous pages (scattered within RAM)

`https://stackoverflow.com/questions/58837677/memory-mapping-in-linux-kernel-use-o`

Kernel address in linux (6)

- the physical memory is splitted into 2 zones
- one zone that would be managed by `kmalloc()`
  - `kmalloc()` allocates memory from the 0 to 896MB within the RAM and not beyond that.
- another zone that is managed by `vmalloc()`
  - `vmalloc()` to allocates memory anywhere from 896MB to 4GB range within the RAM (anywhere in "896MB or higher" range)

https://stackoverflow.com/questions/58837677/memory-mapping-in-linux-kernel-use-o

- 3 different memory managers

1. Physical RAM manager
   mostly keeping track of pages of free physical RAM
2. Virtual space manager
   what is mapped into each virtual address space
   working with fixed size pages
3. Heap memory manager
   allowing a larger area of the virtual address space
   to be split up into arbitrary sized pieces

https://stackoverflow.com/questions/58837677/memory-mapping-in-linux-kernel-use-o

Low / High Memory (1)

- low memory
  - physical memory upto \$<=\$~896MB~
  - 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

Low / High memory (2)

- 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 https://elinu

(1) Logical mapping

- 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

(2) Virtual mapping

- 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

(3) Mapping mechanism

- 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

(4) Two types of kernel addressing

1. Logical Addressing
   - segmented addressing
   - address is formed by base and offset
   - offset in the program is always used
     with the base value in the segment descriptor
2. Linear Addressing : also called virtual address
   - Paging
   - virtual adresses are contigous
   - physical addresses are not contiguous
3. Physical Addressing
   - the actual address on the Main Memory

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

(5) Kernel logical address

- kernel logical addresses use
  normal CPU memory access functions.
- On 32-bit systems,
  only 4GB $= 2^{32}$ 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

(6) Kernel virtual address

- kernel virtual addresses do not necessarily have corresponding logical addresses.
- allocate physical memory with `vmalloc` and get 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

(7) Address from 3GB to 4GB and beyond

- 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 (virtual and physical)
    - `kmalloc()` is used
    - 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

(8) Physically contiguous or not

- to get kernel memory in byte-sized chunks.

	virtual address	physical address
kmalloc()	contiguous	contiguous
vmalloc()	contiguous	not necessarily contiguous

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

(9) kmalloc() and vmalloc()

- 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-

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, to handle the case where
  two or more functions both call kmap on the same page
- kmap can sleep if no mappings are available.

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

Physical Address Extension (PAE)

- PAE sometimes referred to as Page Address Extension, is a memory management feature for the x86 architecture.
- It defines a page table hierarchy of three levels (instead of two), with table entries of 64 bits each instead of 32, allowing these CPUs to directly access a physical address space larger than 4 gigabytes (2^{32} bytes).

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

TOC: Kernel Logical Address

Kernel logical addresses (1)

- normal address space of the kernel
  `kmalloc()`
- virtual addresses are a fixed offset
  from their physical addresses
  virtual 0xc0000000 $\rightarrow$ 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) to physical address
  - __va(x) to logical address
- 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

```
#define __pa(x)  ((unsigned long) (x) - PAGE_OFFSET)
#define __va(x)  ((void *)((unsigned long) (x) + PAGE_OFFSET))
```

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 bottom 1GB of virtual address space, by default
  - the top part of physical memory (3GB - 4GB)
- 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 top part of physical RAM
  is mapped directly into
  kernel logical address space
  - the top part of physical memory (3GB - 4GB)
- 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

TOC: Kernel Virtual Address

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

Kernel virtual addresses (4)

- 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

TOC: User Virtual Address

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

- 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`