Paging On Linux x86-64

Jay Khandkar

26 January 2021

Abstract

In this document we shall take a look at how exactly Linux handles paging by writing a character driver that accepts virtual addresses by means of the `ioctl` interface and walks through the page tables of the process to retrieve the corresponding physical address. This hands-on approach shall demonstrate how to read and understand the kernel source, as well as highlight some important aspects of paging, such as the distinction between page frame numbers and page frames, pages and page frames etc. All aspects of paging described in this document will be specific to the `x86-64` platform.

1 The Virtual Address Space

Firstly, it may be noted that starting with kernel version 4.11, Linux uses five-level page tables as opposed to four-level before. However, for the sake of simplicity, we shall work with four-level tables when describing aspects of paging, and the concept can easily be extended to five-level tables, since it is just one more level of indirection.
Let us now look at how the virtual address space is split in Linux. All 64 bits of the virtual address space are not used. For four-level tables, only the first 48 are used, and bits `48:63` are either all zero (in user space) or all one (in kernel space). To see in detail how this is implemented, let us consult `Documentation/x86/x86_64/mm.rst`:

```
========================================================================================================================
    Start addr    |   Offset   |     End addr     |  Size   | VM area description
========================================================================================================================
                  |            |                  |         |
 0000000000000000 |    0       | 00007fffffffffff |  128 TB | user-space virtual memory, different per mm
__________________|____________|__________________|_________|___________________________________________________________
                  |            |                  |         |
 0000800000000000 | +128    TB | ffff7fffffffffff | ~16M TB | ... huge, almost 64 bits wide hole of non-canonical
                  |            |                  |         |     virtual memory addresses up to the -128 TB
                  |            |                  |         |     starting offset of kernel mappings.
__________________|____________|__________________|_________|___________________________________________________________
                                                            |
                                                            | Kernel-space virtual memory, shared between all processes:
____________________________________________________________|___________________________________________________________
                  |            |                  |         |
 ffff800000000000 | -128    TB | ffff87ffffffffff |    8 TB | ... guard hole, also reserved for hypervisor
 ffff880000000000 | -120    TB | ffff887fffffffff |  0.5 TB | LDT remap for PTI
 ffff888000000000 | -119.5  TB | ffffc87fffffffff |   64 TB | direct mapping of all physical memory (page_offset_base)
 ffffc88000000000 |  -55.5  TB | ffffc8ffffffffff |  0.5 TB | ... unused hole
 ffffc90000000000 |  -55    TB | ffffe8ffffffffff |   32 TB | vmalloc/ioremap space (vmalloc_base)
 ffffe90000000000 |  -23    TB | ffffe9ffffffffff |    1 TB | ... unused hole
 ffffea0000000000 |  -22    TB | ffffeaffffffffff |    1 TB | virtual memory map (vmemmap_base)
 ffffeb0000000000 |  -21    TB | ffffebffffffffff |    1 TB | ... unused hole
 ffffec0000000000 |  -20    TB | fffffbffffffffff |   16 TB | KASAN shadow memory
__________________|____________|__________________|_________|____________________________________________________________
                                                            |
                                                            | Identical layout to the 56-bit one from here on:
____________________________________________________________|____________________________________________________________
                  |            |                  |         |
 fffffc0000000000 |   -4    TB | fffffdffffffffff |    2 TB | ... unused hole
                  |            |                  |         | vaddr_end for KASLR
 fffffe0000000000 |   -2    TB | fffffe7fffffffff |  0.5 TB | cpu_entry_area mapping
 fffffe8000000000 |   -1.5  TB | fffffeffffffffff |  0.5 TB | ... unused hole
 ffffff0000000000 |   -1    TB | ffffff7fffffffff |  0.5 TB | %esp fixup stacks
 ffffff8000000000 | -512    GB | ffffffeeffffffff |  444 GB | ... unused hole
 ffffffef00000000 |  -68    GB | fffffffeffffffff |   64 GB | EFI region mapping space
 ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | ... unused hole
 ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text mapping, mapped to physical address 0
 ffffffff80000000 |-2048    MB |                  |         |
 ffffffffa0000000 |-1536    MB | fffffffffeffffff | 1520 MB | module mapping space
 ffffffffff000000 |  -16    MB |                  |         |
    FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-internal fixmap range, variable size and offset
 ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy vsyscall ABI
 ffffffffffe00000 |   -2    MB | ffffffffffffffff |    2 MB | ... unused hole
__________________|____________|__________________|_________|___________________________________________________________
```

Note that negative addresses such as `-23TB` are absolute addresses in bytes counted down from the top of the 64-bit address space. This makes it slightly easier to visualise the huge 64-bit address space. We can see that user-space virtual memory lies between `0000000000000000` and `00007fffffffffff` and kernel space memory lies above `ffff800000000000`, whence it is clear how the last 16 bits are zero for user-space and 1 for kernel-space: there is a big hole between `0000800000000000` and `ffff7fffffffffff` which is completely unused. Addresses whose bits `48:63` are either all zero or all one are known as `canonical` addresses, and addresses lying in the hole are called `non-canonical` addresses.

The page tables for kernel-space addresses are setup such that there is a 1:1 mapping between kernel virtual addresses and physical addresses, for easy access of physical memory. The macro `PAGE_OFFSET` yields the address where kernel space begins, and it is this value that must be added to any physical address to convert it to a kernel virtual address. Another important macro is `PAGE_SIZE`, which yields the size of a page and is usually `4096` bytes or `4 KiB` for x64 platforms. `PAGE_SHIFT` gives the number of bits `1` must be shifted left to yield `PAGE_SIZE`, and can thus be defined as the logarithm to the base 2 of `PAGE_SIZE`.

2 Page Tables

Let us know look at how exactly virtual addresses are converted into physical ones through page tables, considering specifically the case of four-level page tables. The four different tables are known as

- Page Global Directory
- Page Upper Directory
- Page Middle Directory
- Page Table

Every virtual address (be it kernel or user) is divided into five parts, as shown in figure 1:

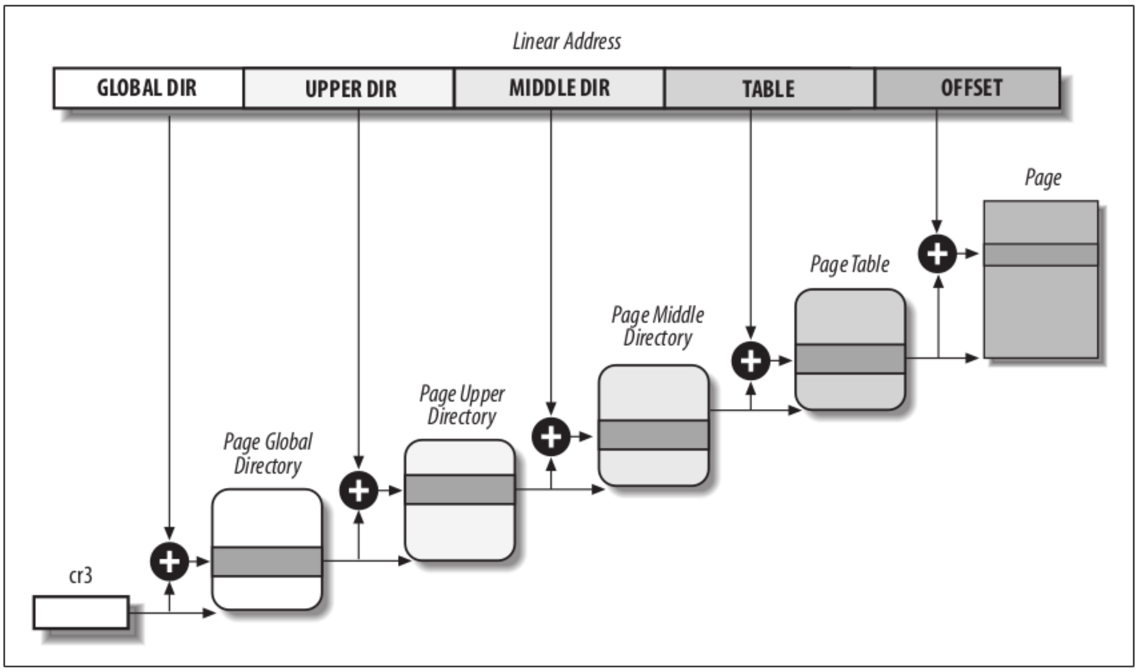

Figure 1: The Linux paging model for four-level page tables

Here, the `OFFSET` part of the address consists of bits `0:11`, the `TABLE` part bits `12:20` and so on. The `GLOBAL DIR` part consists of bits `39:47`. When a process is created, the kernel creates it's corresponding page tables and sets the register `cr3` to point to the base of the Page Global Directory. The `GLOBAL DIR` field within the linear address determines the entry in the Page Global Directory that points to the proper Page Upper Directory. Similarly, the `UPPER DIR` field within the linear address determines the entry in the Page Middle Directory that points to the proper Page Middle Directory. Finally, the Page Table entry gives the physical address of the page itself, and the last 12 bits of the linear address determine the exact address within the page. This process is illustrated in figure 1. A useful macro for calculating the offset is `PAGE_MASK`, which is used to mask all the bits of the `OFFSET` field. This means that `~PAGE_MASK` can be used to calculate the offset from a given virtual address.
Let us now quickly go over how things change for a five-level page table configuration, the one we have to deal with if we're working with kernels later than `4.11`. Now, bits `0:56` of the 64 bit address space are used, and the extra page table is simply called `p4d`. Correspondingly, there would be one more table and one more level of indirection in figure 1.
Even more configurations for different 64 bit architectures are given in Table 1

Platform Name	Page Size	Number of address bits used	Number of paging levels	Linear address splitting
alpha	8KiB	43	3	10 + 10 + 10 + 13
ia64	4KiB	39	3	9 + 9 + 9 + 12
ppc64	4KiB	41	3	10 + 10 + 9 + 12
sh64	4KiB	41	3	10 + 10 + 9 + 12
x86-64 [4 level]	4KiB	48	4	9 + 9 + 9 + 9 + 12
x86-64 [5 level]	4KiB	57	5	9 + 9 + 9 + 9 + 9 + 12

Table 1: Paging levels in some 64 bit architectures

3 Our Character Driver - VTP

We are now in a position to start writing our character driver. Our driver, called `vtp` will do the following:

- Accept, through `ioctl`, a user-space virtual address.
- Walk through the `current` process' page tables to find the corresponding physical address
- Print the physical address of the page frame, and the physical address itself to the kernel log buffer

We shall allocate a device number for our char device dynamically using `alloc_chrdev_region`. We can then write a small shell script to find the allocated number through `/proc/devices` and create the corresponding file using `mknod`. The `init` function for our driver is shown in listing 1.

```c
#include <linux/kernel.h>        /* printk() */
#include <linux/slab.h>          /* kmalloc() */
#include <linux/fs.h>            /* everything... */
#include <linux/errno.h>         /* error codes */
#include <linux/cdev.h>    /* char device registration*/
#include <linux/types.h>
#include <linux/fcntl.h>         /* O_ACCMODE */
#include <linux/mm_types.h>      /* struct page and struct mm_struct*/
#include <asm/page.h>            /*pgd_t, pte_t, __va etc.*/
#include <linux/pgtable.h>       /*pgd_offset, pud_offset etc*/
#include <asm/pgtable_types.h>   /*PTE_PFN_MASK*/
#include <linux/highmem.h>
#include <asm/io.h>

MODULE_LICENSE("Dual BSD/GPL");

int vtp_major = 0;    /*we shall allocate dynamically*/
int vtp_minor = 0;
int nr_vtp_devs = 1;    /*just one device*/

struct cdev *vtp_cdev;    /*our character device*/
const struct file_operations vtp_fops = {
   .owner =    THIS_MODULE,
   .unlocked_ioctl = vtp_ioctl,   /*we'll only use ioctl*/
};
int vtp_init(void)
{
  int result, err;
  dev_t dev = 0;

  result = alloc_chrdev_region(&dev, vtp_minor, nr_vtp_devs,
              "vtp");
  vtp_major = MAJOR(dev);

  if (result < 0){
    printk(KERN_WARNING "vtp: can't allocate device number\n");
    return result;
  }

  vtp_cdev = cdev_alloc();
  vtp_cdev->ops = &vtp_fops;
  vtp_cdev->owner = THIS_MODULE;
  err = cdev_add(vtp_cdev, dev, 1);
  if (err)
    printk(KERN_NOTICE "couldn't add cdev: error %d", err);
  return 0;
}
```

Listing 1: The init function

When writing the script to create the char device, we have to be careful since the script has been executed by `superuser`. Here, we choose to give the group `sudo` read and write access to the device. The script to load the module and create the char device is shown in listing 2.

```
#!/bin/sh

module="vtp"
device="vtp"
mode="664"  #user, group [rw] others [r]

#insert module
/sbin/insmod ./$module.ko $* || exit 1

#remove devices if already exist
rm -f /dev/${device}[0]

#find the major number
major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)

mknod /dev/${device}0 c $major 0

#now give appropriate permissions to other users, since we have been
#invoked by superuser
group="sudo"

chgrp $group /dev/${device}0
chmod $mode /dev/${device}0
```

Listing 2: The vtp_load script

Let us now move on to our `ioctl` implementation, where we'll actually walk the page tables. The various page table entries are represented in the kernel by the datatypes `pgd_t`, `p4d_t`, `pud_t`, `pmd_t` and `pte_t`. The address to the Page Global Directory of the current process is stored in the `struct mm_struct` of the `current` process. We can get the correct PGD entry corresponding to our address using the macro `pgd_offset` defined in `linux/pgtable.h`. This macro takes a `struct mm_struct` and a virtual address as a parameter and returns a pointer to the corresponding PGD entry.
There are four other macros, `p4d_offset`, `pud_offset`, `pmd_offset` and `pte_offset_kernel` which serve the same purpose as that of `pgd_offset`. These macros are applied sequentially one after the other, the result of one being the parameter to the next, essentially walking through the page tables. After we have applied `pte_offset_kernel`, what we end up with is the physical address of the page frame corresponding to our virtual address. What remains is to add the last 12 bits of our virtual address (the offset part), and we will have obtained the exact physical address. One last thing to take care of is the fact that `pte_offset_kernel` will give us an address containing various bits for flags such as access rights, page size etc. So, we will have to use `PTE_PFN_MASK`, defined in `asm/pgtable_types.h` to mask off those bits and obtain the actuall physical address. The code for our `ioctl` implementation is shown in listing 3.

```
long vtp_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
  char c;

  pgd_t *pgd;
  pte_t *ptep;
  pud_t *pud;
  p4d_t *p4d;
  pmd_t *pmd;
  char *addr, *pf_addr;

  struct page *page = NULL;
  struct mm_struct *mm = current->mm;

  pgd = pgd_offset(mm, arg);
  if (pgd_none(*pgd) || pgd_bad(*pgd))
    goto out;
  printk(KERN_NOTICE "Valid pgd\n");

  p4d = p4d_offset(pgd, arg);
  if (p4d_none(*p4d) || p4d_bad(*p4d))
    goto out;
  printk(KERN_NOTICE "Valid p4d\n");

  pud = pud_offset(p4d, arg);
  if (pud_none(*pud) || pud_bad(*pud))
```

```
    goto out;
  printk(KERN_NOTICE "Valid pud\n");

  pmd = pmd_offset(pud, arg);
  if (pmd_none(*pmd) || pmd_bad(*pmd))
    goto out;
  printk(KERN_NOTICE "Valid pmd\n");

  ptep = pte_offset_kernel(pmd, arg);
  if(!ptep)
    goto out;

  page = pte_page(*ptep);
  pf_addr = (char *)((unsigned long)pte_val(*ptep) & PTE_PFN_MASK);
  addr = pf_addr + (arg & ~PAGE_MASK);
  c = *((char *) __va(addr));
  printk(KERN_INFO "the physical address is 0x%px\n", (void *)addr);
  printk(KERN_INFO "the physical page frame address is 0x%px\n", (void *)pf_addr);
  printk(KERN_INFO "and the kernel virt addr is 0x%px\n", (void*)__va(addr));
  printk(KERN_INFO "the byte there is 0x%x\n", c);
  pte_unmap(ptep);

  return 0;

  out:
  printk(KERN_INFO "couldn't walk page tables\n");
  return -1;
}
```

Listing 3: Our ioctl function