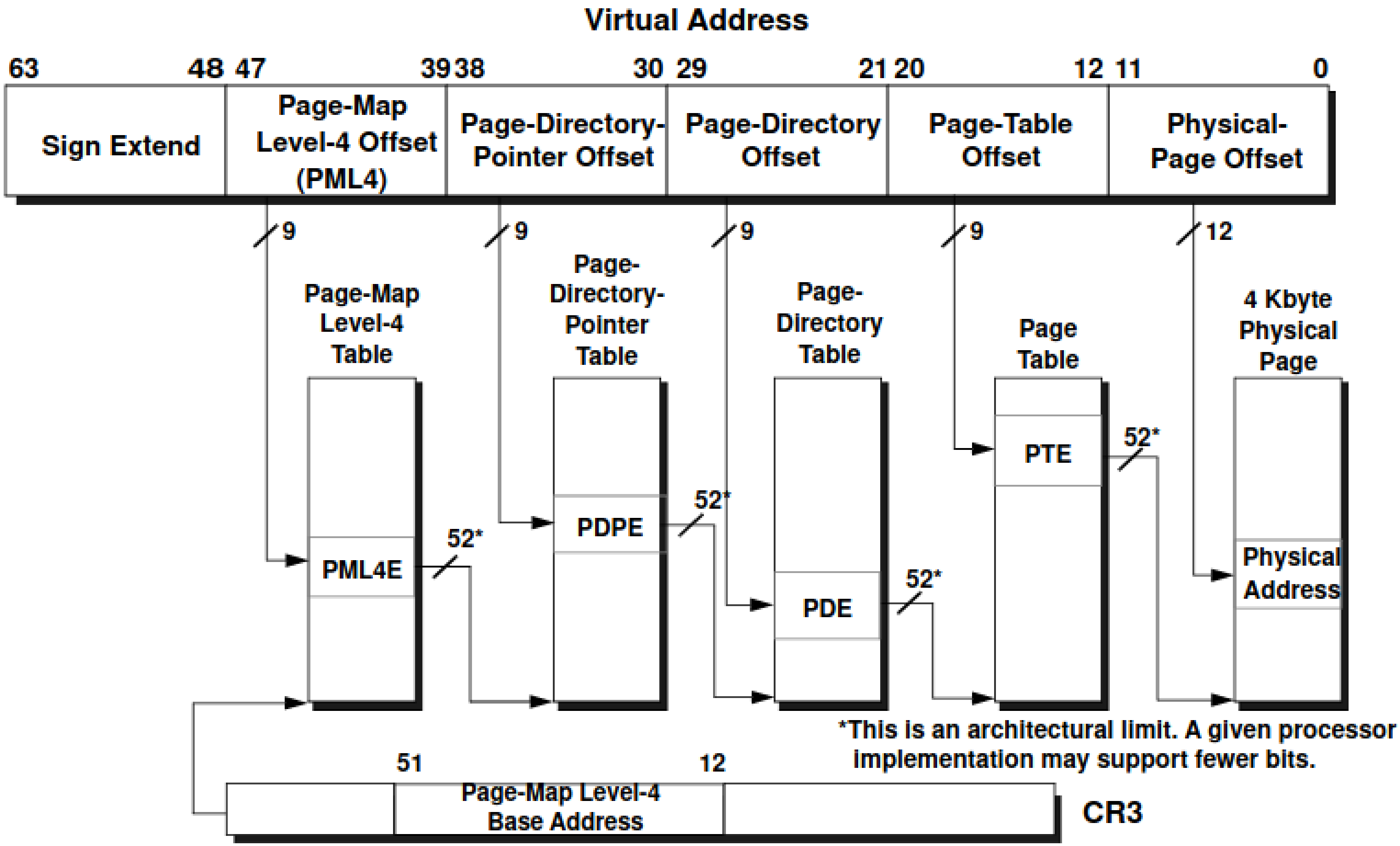
# CSE 511: Operating Systems Design

## Lectures 5,6

Page Tables

Kernel Booting

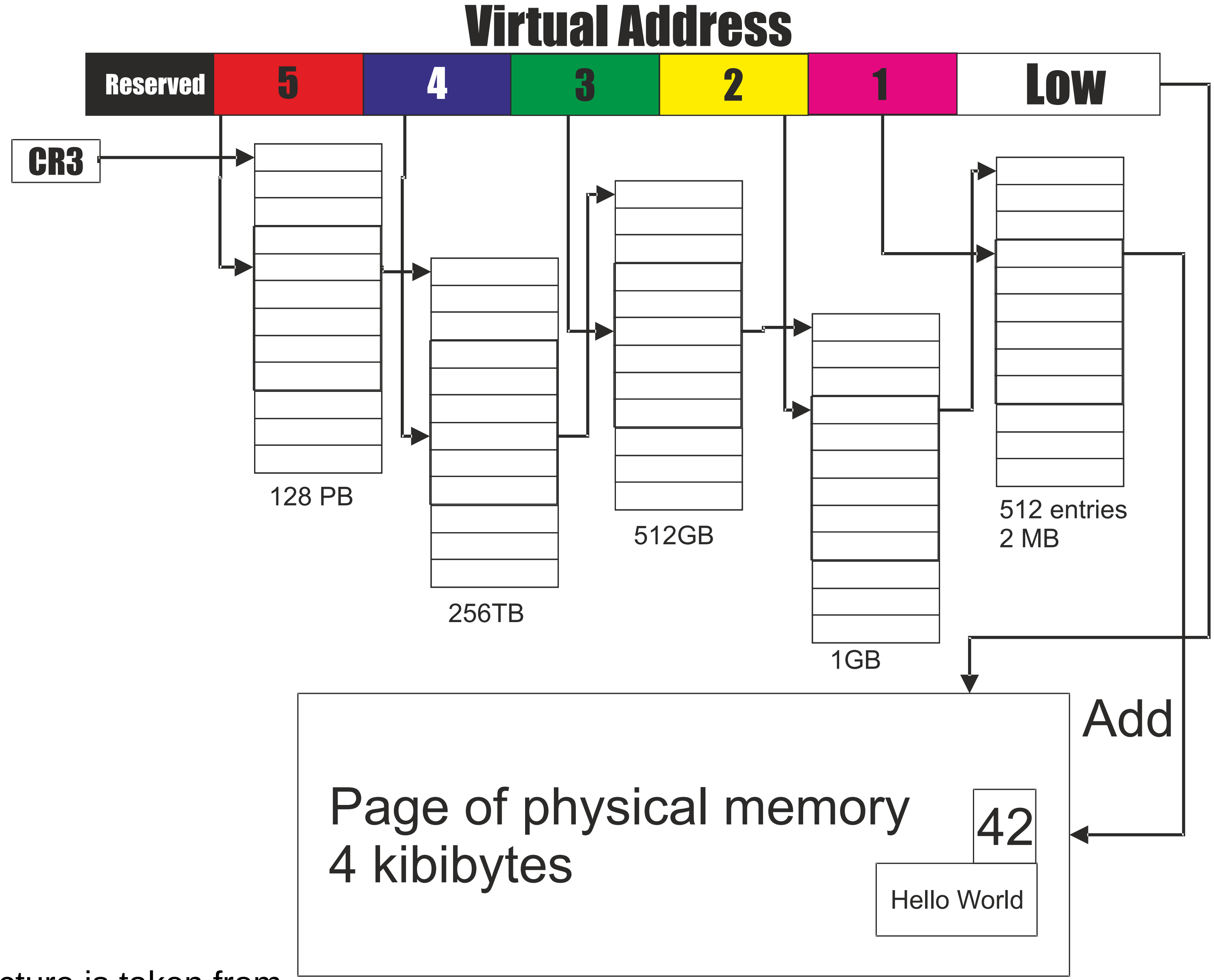
# Traditional 48-bit paging (4 levels)



*This is an architectural limit. A given processor implementation may support fewer bits.

* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

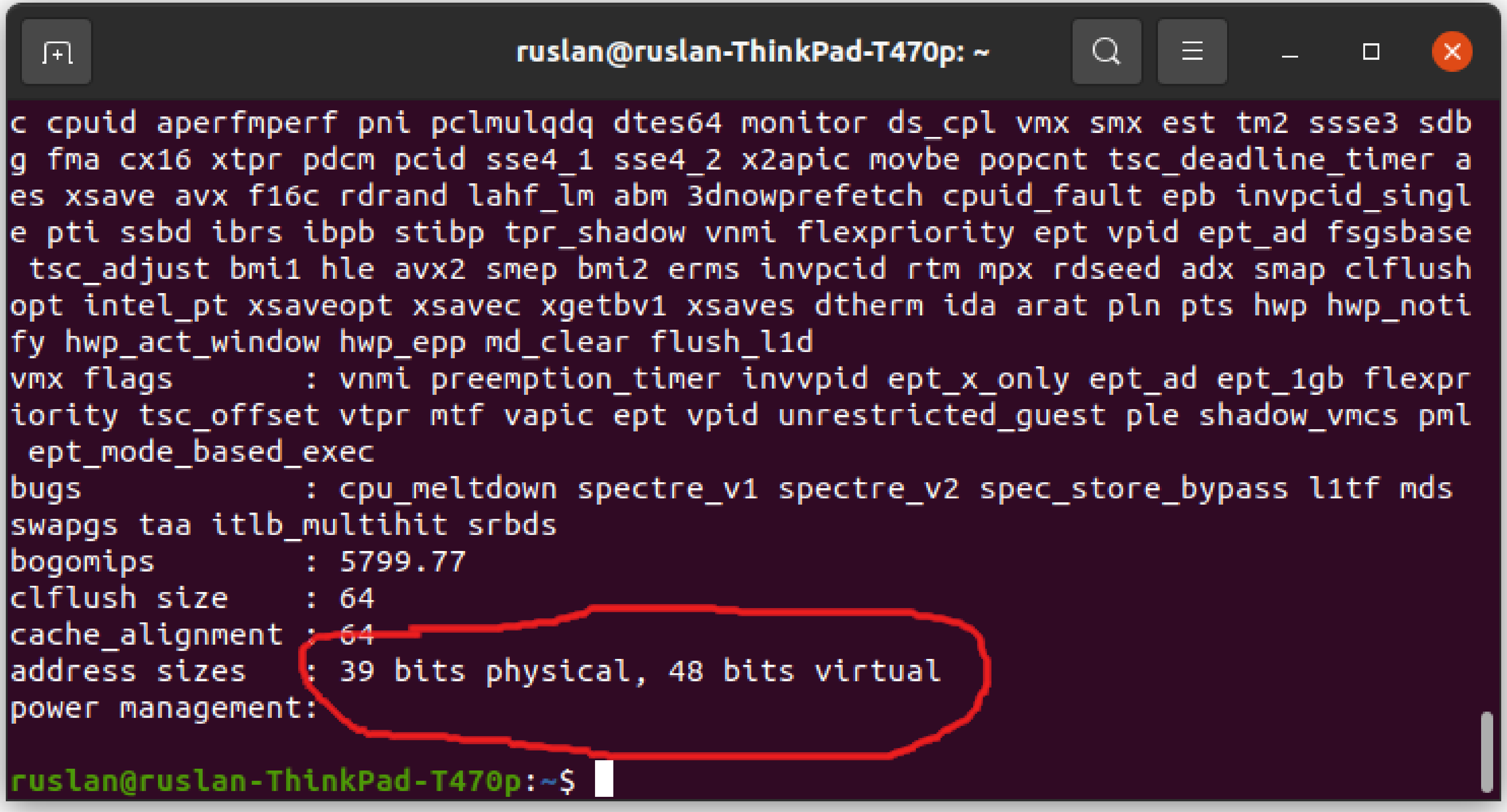
# Extension: 57-bit paging (5 levels)



* The picture is taken from https://en.wikipedia.org/wiki/Intel_5-level_paging

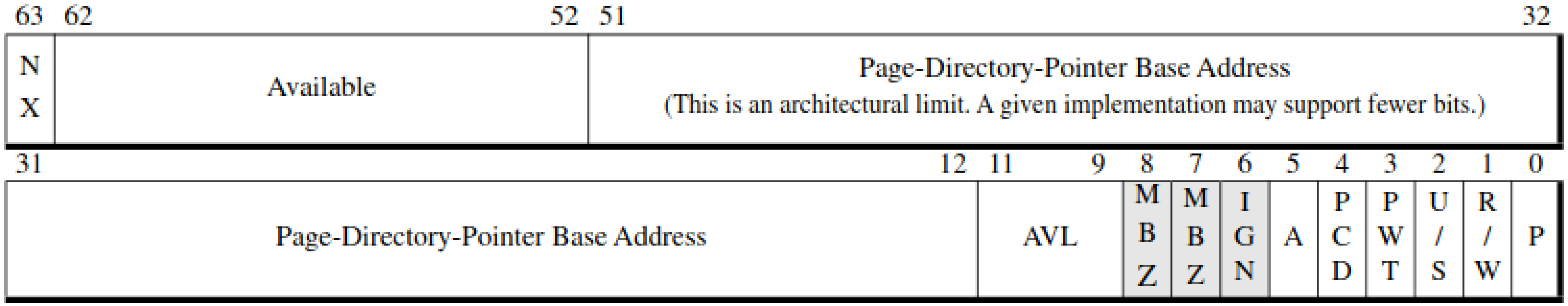
# Page Table

```
cat /proc/cpuinfo
```



```
c cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdb
g fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer a
es xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_singl
e pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase
 tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdseed adx smap clflush
opt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_noti
fy hwp_act_window hwp_epp md_clear flush_l1d
vmx flags       : vnmi preemption_timer invvpid ept_x_only ept_ad ept_1gb flexpr
iority tsc_offset vtpr mtf vapic ept vpid unrestricted_guest ple shadow_vmcs pml
 ept_mode_based_exec
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs taa itlb_multihit srbds
bogomips        : 5799.77
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

ruslan@ruslan-ThinkPad-T470p:~$ 
```

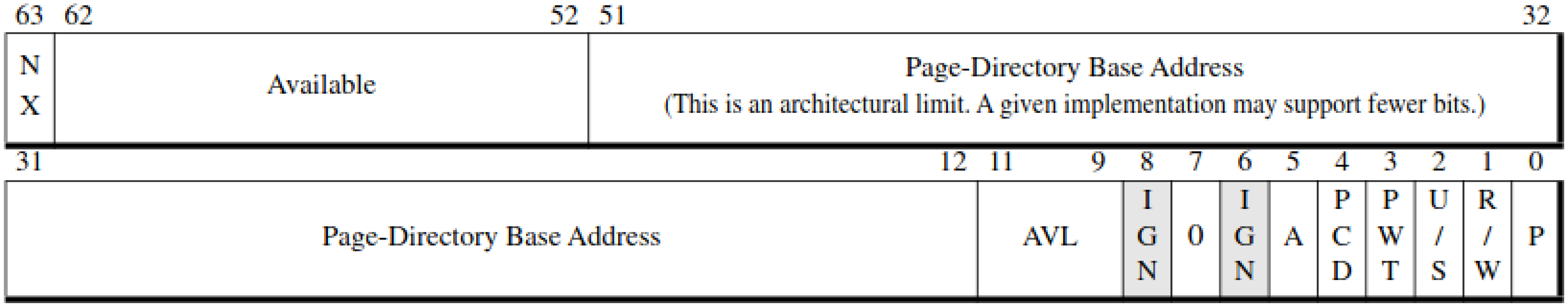
# Page Table



| 63 | 62 … 52 | 51 … 32 |
|---|---|---|
| NX | Available | Page-Directory-Pointer Base Address<br>(This is an architectural limit. A given implementation may support fewer bits.) |

| 31 … 12 | 11 … 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Page-Directory-Pointer Base Address | AVL | MBZ | MBZ | IGN | A | PCD | PWT | U/S | R/W | P |

**Figure 5-18. 4-Kbyte PML4E—Long Mode**



| 63 | 62 … 52 | 51 … 32 |
|---|---|---|
| NX | Available | Page-Directory Base Address<br>(This is an architectural limit. A given implementation may support fewer bits.) |

| 31 … 12 | 11 … 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Page-Directory Base Address | AVL | IGN | 0 | IGN | A | PCD | PWT | U/S | R/W | P |

**Figure 5-19. 4-Kbyte PDPE—Long Mode**

* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

# Page Table

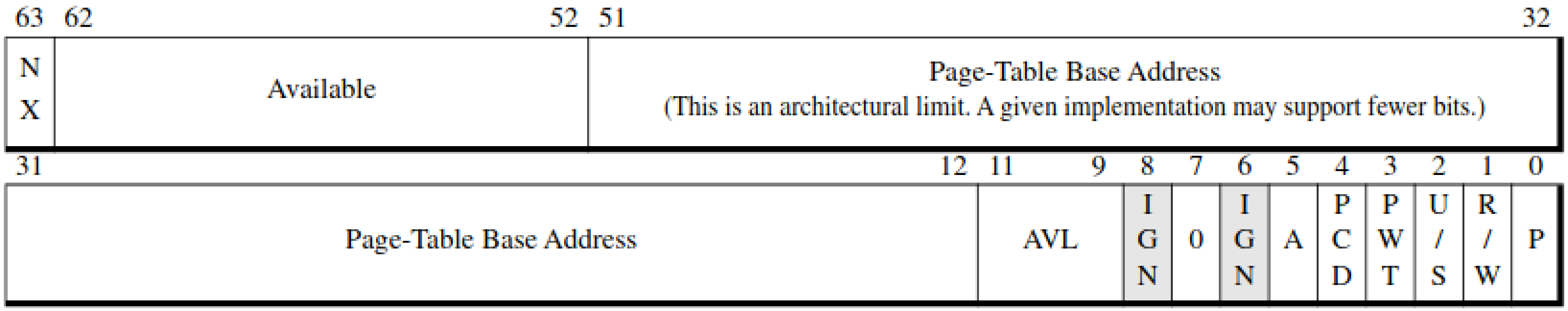


**Figure 5-20. 4-Kbyte PDE—Long Mode**

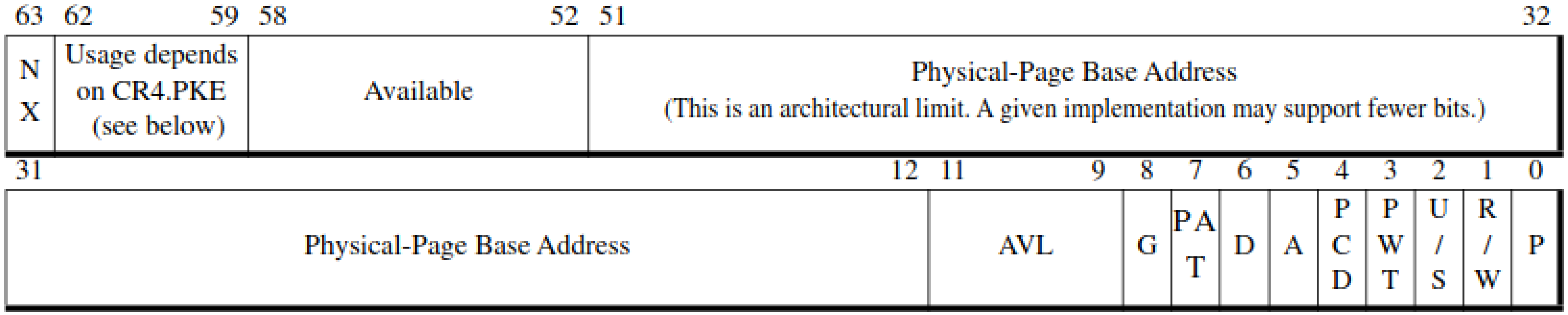

63
62
59
58
52
51
32
N X
Usage depends on CR4.PKE (see below)
Available
Physical-Page Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12
11
9
8
7
6
5
4
3
2
1
0
Physical-Page Base Address
AVL
G
PAT
D
A
PCD
PWT
U/S
R/W
P

**Figure 5-21. 4-Kbyte PTE—Long Mode**

* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

# Example: PTE

```
typedef unsigned long long u64;

struct page_pte {
    u64 present:1;       // Bit P
    u64 writable:1;      // Bit R/W
    u64 user_mode:1;     // Bit U/S
    ...
    u64 page_address:40; // 40+12 = 52-bit physical address (max)
    U64 avail:7;         // reserved, should be 0
    u64 pke:4;           // no MPK/PKE, should be 0
    u64 nonexecute:1;
};
```

# Example: PTE

```
typedef unsigned long long u64;

struct page_pte {
    u64 present:1;       // Bit P
    u64 writable:1;      // Bit R/W
    u64 user_mode:1;     // Bit U/S
    ...
    u64 page_address:40; // 40+12 = 52-bit physical address (max)
    U64 avail:7;         // reserved, should be 0
    u64 pke:4;           // no MPK/PKE, should be 0
    u64 nonexecute:1;
};

// _page_ address, i.e. memory_address / 4096 (or 2^12)
p->page_address = 0xZZZZZZZZZZULL; // ULL is unsigned long long
p->writable = 1;
p->present = 1;
p->user_mode = 0;
...
p->avail = 0;
p->pke = 0;
p->nonexecute = 0;
```

# Example: Initializing 4 GB

For 4GB, we reference 1048576 physical pages
Using 2048 pages for PTEs, 4 pages for PDEs, 1 for PDPE, 1 for PMLE4E
Total: 2054 pages = 8413184 bytes, align at the 4096 boundary!

**PTE:**

```
struct page_pte *p; … // Each entry is 8 bytes
For (int i = 0; i < 1048576; i++) { // 1048576/512 = 2048 pages
   p[i] = … // physical pages 0,1,2,3,…,1048575 (absolute address)
}
```

**PDE:**

```
struct page_pde *pd = (struct page_pde *) (p + 1048576);
for (int j = 0; j < 2048; j++) { // 2048/512 = 4 pages
   struct page_pte *start_pte = p + 512 * j;
   page_addr = (u64) start_pte >> 12; // we record the page address
   …
}
```

**PDPE:**

Reference 4 PDEs (1 page), everything else is empty

**PMLE4E:**

Just one reference to PDPE; everything else is empty

# Example: Initializing 4 GB

- Do not use more than 1GB RAM! (qemu: 1024)
    - The Video RAM is after that (but before 4GB)

# Aligning Pages

```
Allocate more space: e.g., 8413184 + 4095

Align the allocated ‘base’:
(void *) (((unsigned long long) base + 4095) & (~4095ULL))
```

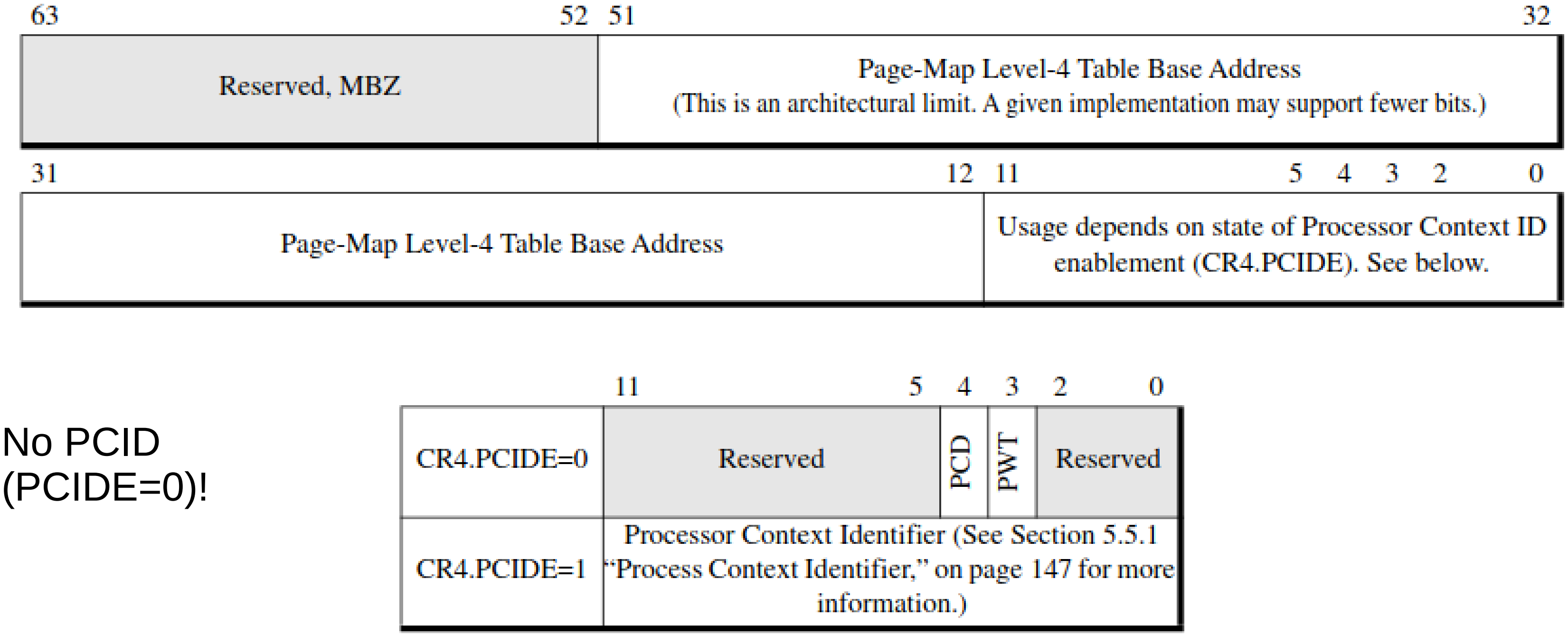
- Why long long?
  - The kernel uses 64-bit ‘long’ due to System V’s ABI (aka the LP64 model)
  - The boot loader uses 32-bit ‘long’ due to EFI/Microsoft’s ABI (aka the LLP64 model)
  - ‘int’ is 32 bit and ‘long long’ is 64 bit in either case

# Loading Page Table

```
void write_cr3(unsigned long long cr3_value)
{
    asm volatile ("mov %0, %%cr3"
                 :
                 : "r" (cr3_value)
                 : "memory");
}
```

# Loading Page Table

```
void write_cr3(unsigned long long cr3_value)
{
    asm volatile ("mov %0, %%cr3"
                 :
                 : "r" (cr3_value)
                 : "memory");
}
```



| 63 52 | 51 32 |
|---|---|
| Reserved, MBZ | Page-Map Level-4 Table Base Address<br>(This is an architectural limit. A given implementation may support fewer bits.) |

| 31 12 | 11 5 4 3 2 0 |
|---|---|
| Page-Map Level-4 Table Base Address | Usage depends on state of Processor Context ID enablement (CR4.PCIDE). See below. |

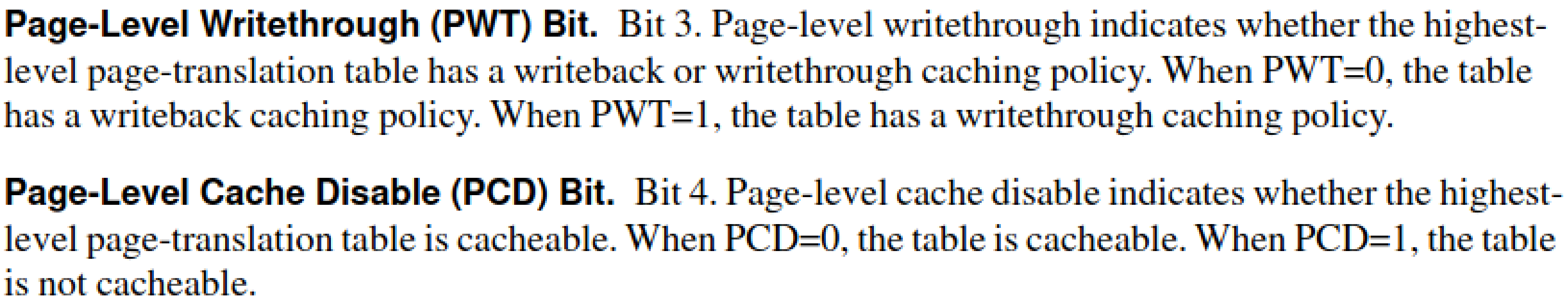
No PCID (PCIDE=0)!

| | 11 5 | 4 | 3 | 2 0 |
|---|---|---|---|---|
| CR4.PCIDE=0 | Reserved | PCD | PWT | Reserved |
| CR4.PCIDE=1 | Processor Context Identifier (See Section 5.5.1 "Process Context Identifier," on page 147 for more information.) | | | |

**Figure 3-6. Control Register 3 (CR3)—Long Mode**

* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

# Loading Page Table



**Page-Level Writethrough (PWT) Bit.** Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

**Page-Level Cache Disable (PCD) Bit.** Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

```
PCD=0, PWT=0
```

# OS Boot Process

- We have to load the main kernel image
  - An executable file (e.g., ELF) which is loaded at some **fixed** *virtual* address (e.g., 0x100000, right after the first 1MB of legacy BIOS/DOS memory)
  - Relocatable kernels are also possible
    - Kernel-address space layout randomization (KASLR) to enhance security
- The kernel image may load additional modules during the boot process
  - Typically relocatable executables

# “Chicken-and-egg” Boot Problem

- We need to load file system and storage (or network) drivers to load kernel modules
  - But file system and storage (or network) drivers are often compiled as modules
  - We often cannot use firmware drivers once we left the OS boot loader
- What do we do?
  - The OS boot loader must preload critical (but not all!) modules

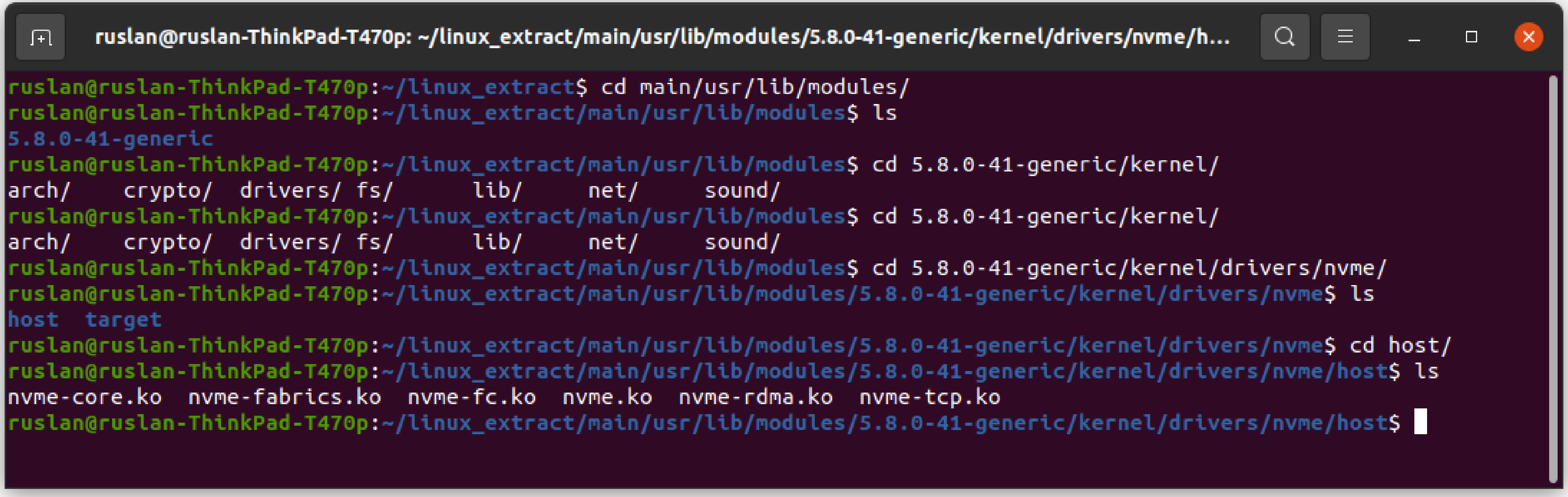
# Example: Linux Protocol

- The Linux kernel executable image is compressed in the “/boot/vmlinuz” file
- All critical modules are placed in “initrd.img”
  - It also has other files (e.g., configuration, microcode, etc)
- For example, in GRUB (/boot/grub/grub.cfg):

```
    linux   /boot/vmlinuz-5.8.0-41-generic root=UUID=00000000-0000-0000-0000-000000000000 ro  quiet splash $vt_handoff
    initrd  /boot/initrd.img-5.8.0-41-generic
```

# What is inside initrd.img?

```
$ mkdir linux_extract
$ cd linux_extract/
$ unmkinitramfs -v /boot/initrd.img-<version>-generic .
```


ruslan@ruslan-ThinkPad-T470p: ~/linux_extract/main/usr/lib/modules/5.8.0-41-generic/kernel/drivers/nvme/h...

```
ruslan@ruslan-ThinkPad-T470p:~/linux_extract$ cd main/usr/lib/modules/
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules$ ls
5.8.0-41-generic
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules$ cd 5.8.0-41-generic/kernel/
arch/    crypto/  drivers/ fs/      lib/     net/     sound/
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules$ cd 5.8.0-41-generic/kernel/
arch/    crypto/  drivers/ fs/      lib/     net/     sound/
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules$ cd 5.8.0-41-generic/kernel/drivers/nvme/
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules/5.8.0-41-generic/kernel/drivers/nvme$ ls
host  target
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules/5.8.0-41-generic/kernel/drivers/nvme$ cd host/
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules/5.8.0-41-generic/kernel/drivers/nvme/host$ ls
nvme-core.ko  nvme-fabrics.ko  nvme-fc.ko  nvme.ko  nvme-rdma.ko  nvme-tcp.ko
ruslan@ruslan-ThinkPad-T470p:~/linux_extract/main/usr/lib/modules/5.8.0-41-generic/kernel/drivers/nvme/host$ 
```

# Example: GRUB’s “Multiboot” Protocol

- Can be used by many OS kernels if they implement this protocol (Examples: NetBSD, Xen, ...)
- For example for Xen (/boot/grub/grub.cfg)
  - We load the “kernel”, called xen.gz (Xen)
  - We load modules
    - vmlinuz is a Xen “module”, i.e., an OS kernel
    - initrd is another module (but for Linux)

```
    multiboot2  /boot/xen.gz placeholder   ${xen_rm_opts}
    echo    'Loading Linux 5.8.0-41-generic ...'
    module2 /boot/vmlinuz-5.8.0-41-generic placeholder root=UUID=00000000-0000-0000-0000-000000000000 ro  quiet splash
    echo    'Loading initial ramdisk ...'
    module2 --nounzip   /boot/initrd.img-5.8.0-41-generic
```

# State of the system

- Interrupts are disabled
  - Need to be set up
- Page tables can be very basic (“early boot”)
  - The kernel will set up its own page tables
- Other processors need to be brought up
  - Not clear if EFI_MP_SERVICES_PROTOCOL will change that in the future
- Little or no possibility to print anything right away
  - Some “early” printk() can still be available

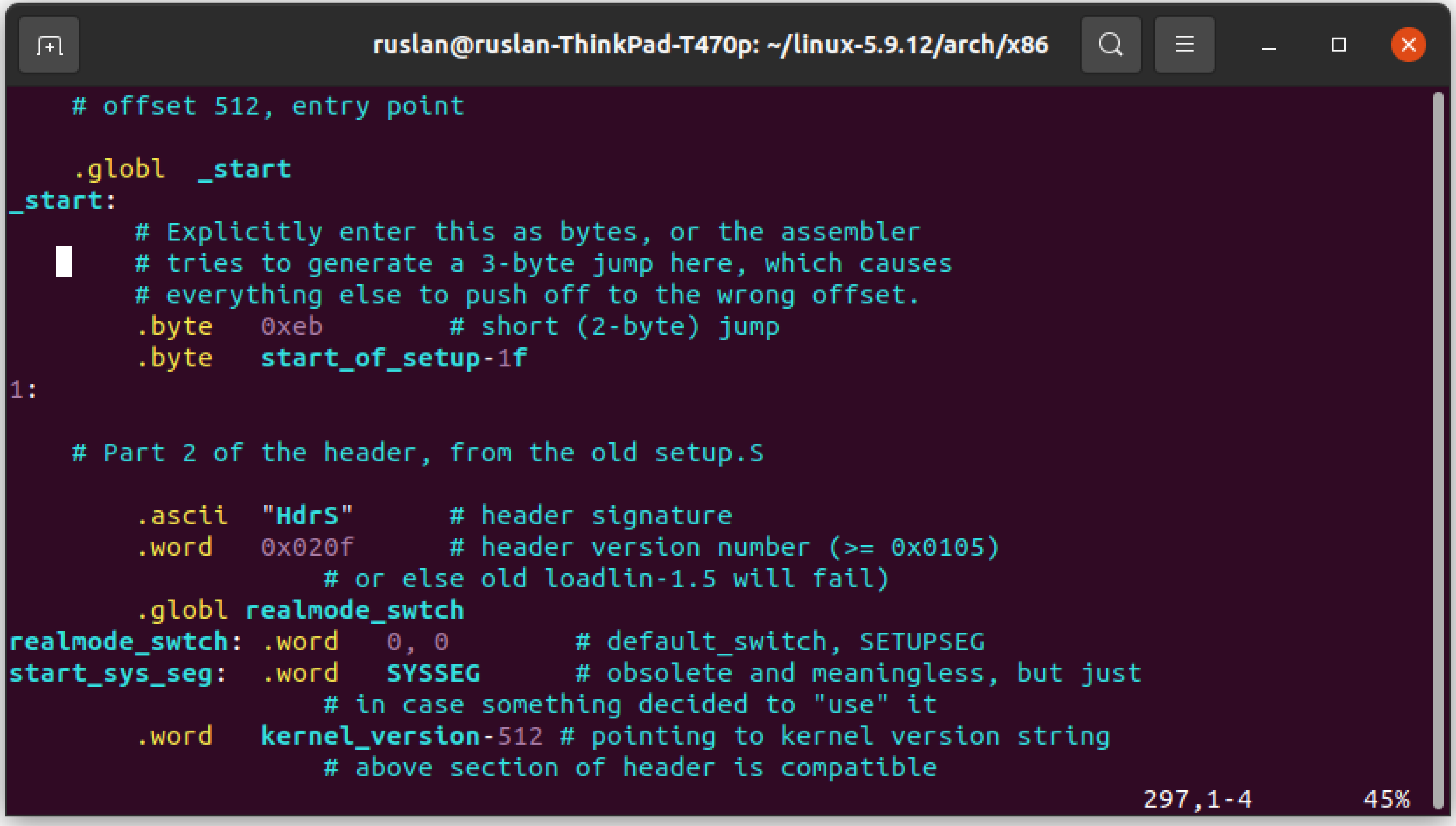
# Linux Boot Process

- For BIOS, there was no recourse
- For UEFI, we can specify CONFIG_EFI_STUB
  - The boot loader is embedded in the Linux kernel
  - ExitBootServices() is called by Linux
  - No support for Linux file systems, we have to place files on the EFI partition (even if the kernel has built-in drivers)
- EFI handover (new protocol)

```
linuxefi  /boot/efi/EFI/vmlinuz.efi
initrdefi /boot/efi/EFI/initramfs.img
```

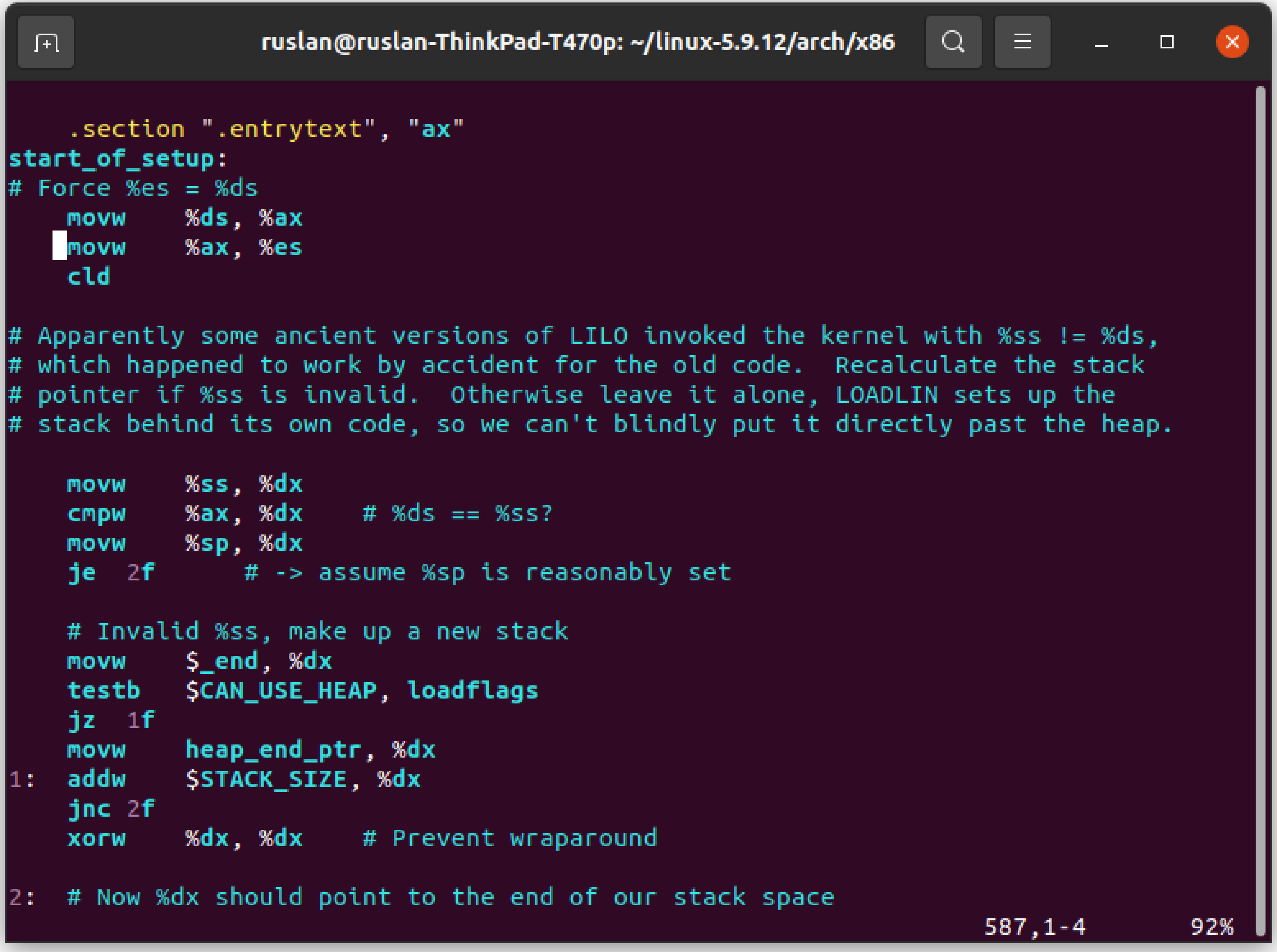
# Linux Boot Process

- arch/x86/boot/header.S



```
    # offset 512, entry point

    .globl  _start
_start:
        # Explicitly enter this as bytes, or the assembler
        # tries to generate a 3-byte jump here, which causes
        # everything else to push off to the wrong offset.
        .byte   0xeb        # short (2-byte) jump
        .byte   start_of_setup-1f
1:

    # Part 2 of the header, from the old setup.S

        .ascii  "HdrS"      # header signature
        .word   0x020f      # header version number (>= 0x0105)
                    # or else old loadlin-1.5 will fail)
        .globl realmode_swtch
realmode_swtch: .word   0, 0        # default_switch, SETUPSEG
start_sys_seg:  .word   SYSSEG      # obsolete and meaningless, but just
                    # in case something decided to "use" it
        .word   kernel_version-512 # pointing to kernel version string
                    # above section of header is compatible
                                                                        297,1-4       45%
```

# Linux Boot Process

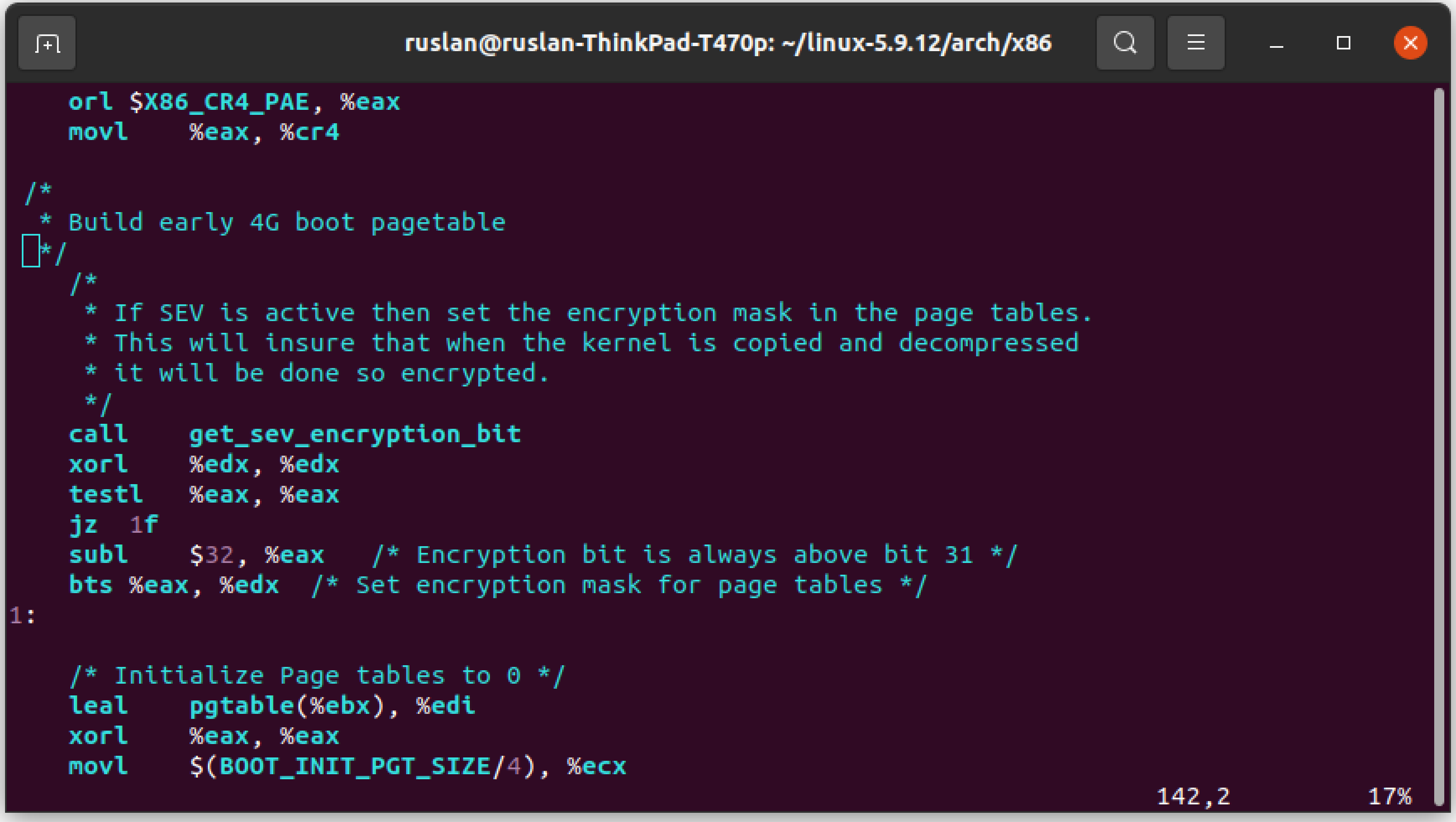


ruslan@ruslan-ThinkPad-T470p: ~/linux-5.9.12/arch/x86

```
    .section ".entrytext", "ax"
start_of_setup:
# Force %es = %ds
    movw    %ds, %ax
    movw    %ax, %es
    cld

# Apparently some ancient versions of LILO invoked the kernel with %ss != %ds,
# which happened to work by accident for the old code.  Recalculate the stack
# pointer if %ss is invalid.  Otherwise leave it alone, LOADLIN sets up the
# stack behind its own code, so we can't blindly put it directly past the heap.

    movw    %ss, %dx
    cmpw    %ax, %dx    # %ds == %ss?
    movw    %sp, %dx
    je  2f      # -> assume %sp is reasonably set

    # Invalid %ss, make up a new stack
    movw    $_end, %dx
    testb   $CAN_USE_HEAP, loadflags
    jz  1f
    movw    heap_end_ptr, %dx
1:  addw    $STACK_SIZE, %dx
    jnc 2f
    xorw    %dx, %dx    # Prevent wraparound

2:  # Now %dx should point to the end of our stack space
                                                              587,1-4       92%
```

# Linux Boot Process

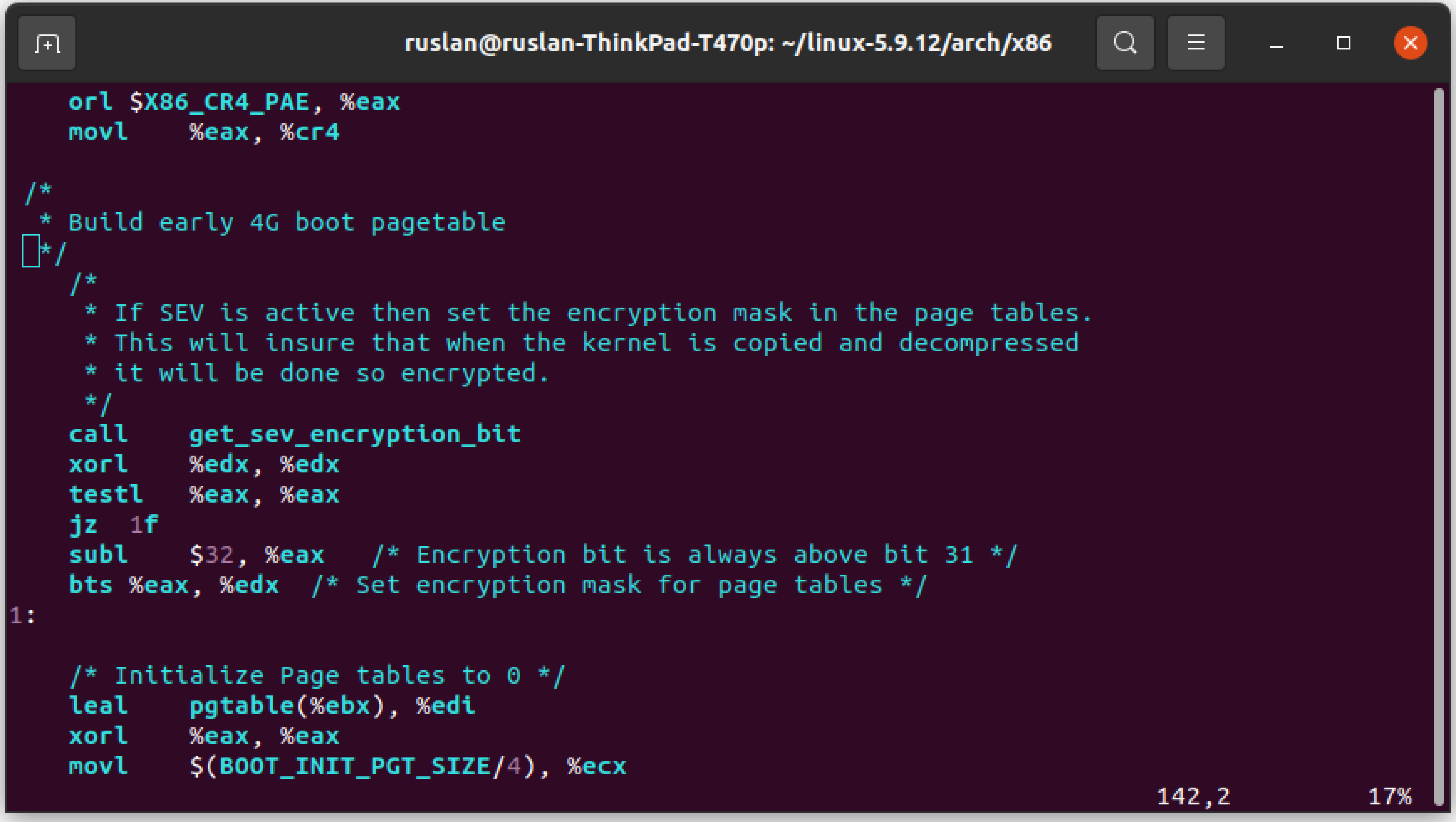
- boot/compressed/head_64.S



```
	orl $X86_CR4_PAE, %eax
	movl	%eax, %cr4

 /*
  * Build early 4G boot pagetable
  */
	/*
	 * If SEV is active then set the encryption mask in the page tables.
	 * This will insure that when the kernel is copied and decompressed
	 * it will be done so encrypted.
	 */
	call	get_sev_encryption_bit
	xorl	%edx, %edx
	testl	%eax, %eax
	jz	1f
	subl	$32, %eax	/* Encryption bit is always above bit 31 */
	bts	%eax, %edx	/* Set encryption mask for page tables */
1:

	/* Initialize Page tables to 0 */
	leal	pgtable(%ebx), %edi
	xorl	%eax, %eax
	movl	$(BOOT_INIT_PGT_SIZE/4), %ecx
                                                            142,2         17%
```

# Linux Boot Process

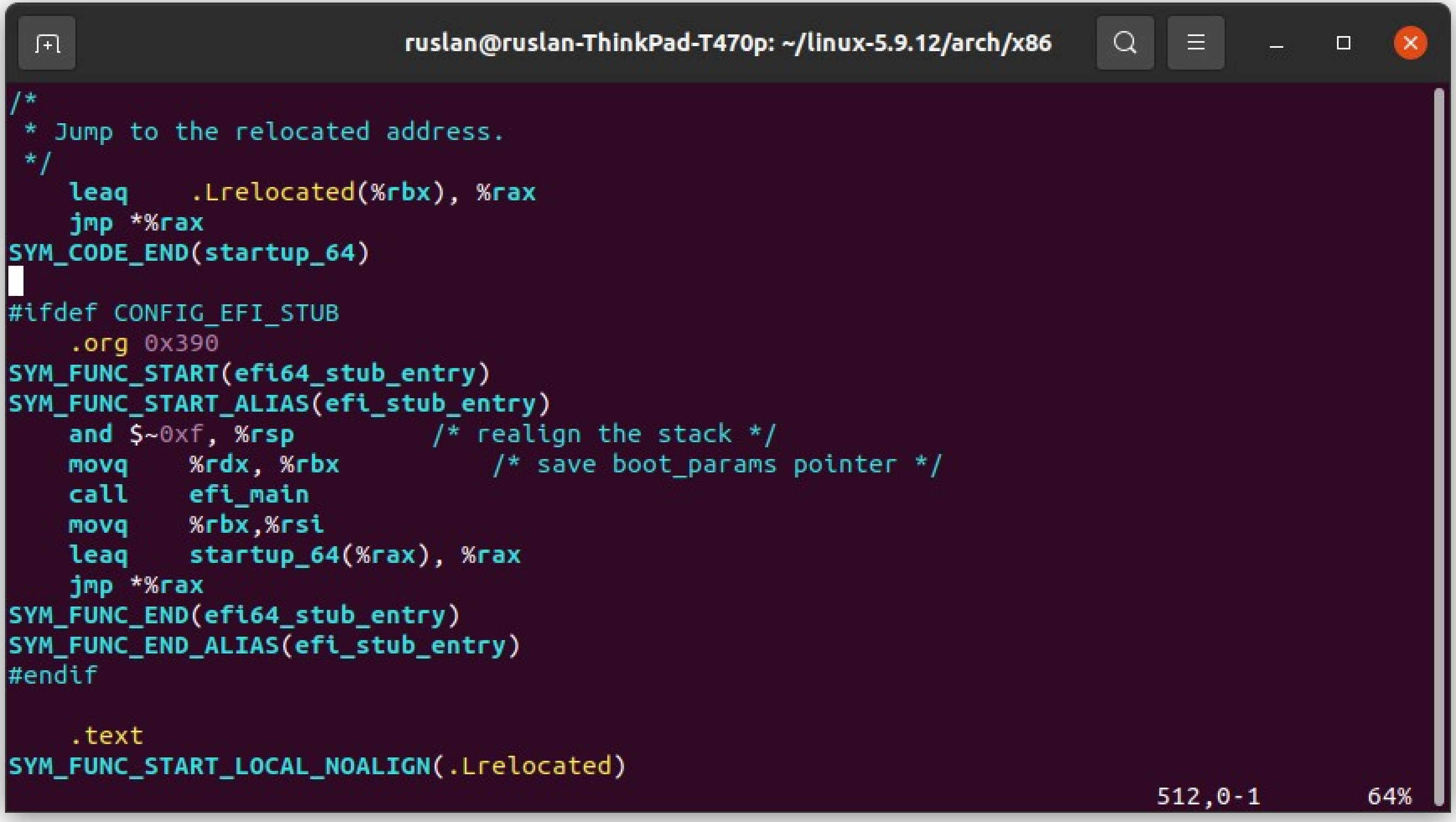
- boot/compressed/head_64.S



```
    orl $X86_CR4_PAE, %eax
    movl    %eax, %cr4

 /*
  * Build early 4G boot pagetable
  */
    /*
     * If SEV is active then set the encryption mask in the page tables.
     * This will insure that when the kernel is copied and decompressed
     * it will be done so encrypted.
     */
    call    get_sev_encryption_bit
    xorl    %edx, %edx
    testl   %eax, %eax
    jz  1f
    subl    $32, %eax   /* Encryption bit is always above bit 31 */
    bts %eax, %edx  /* Set encryption mask for page tables */
1:

    /* Initialize Page tables to 0 */
    leal    pgtable(%ebx), %edi
    xorl    %eax, %eax
    movl    $(BOOT_INIT_PGT_SIZE/4), %ecx
                                                                      142,2          17%
```

# Linux Boot Process (EFI)

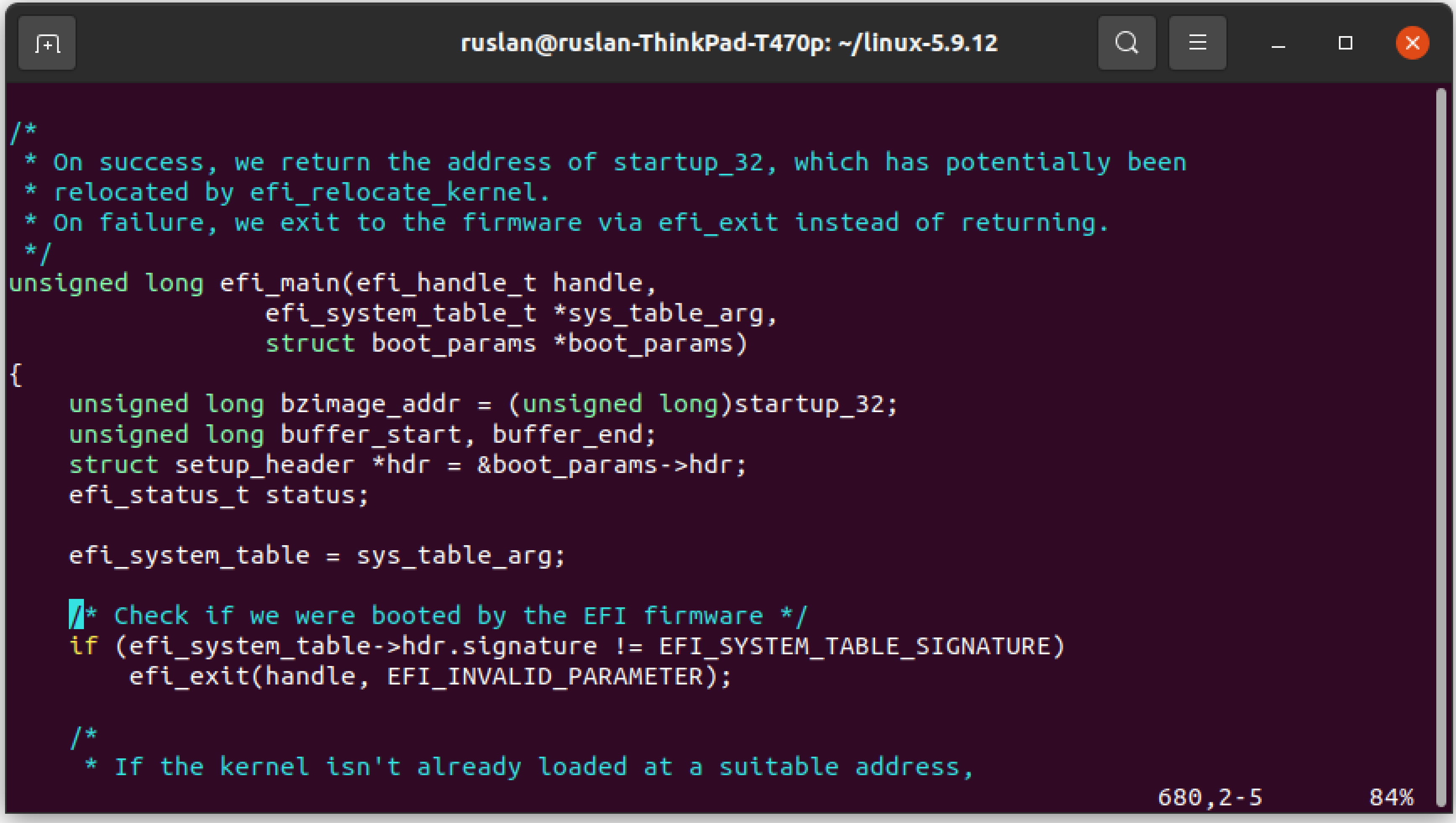
- boot/compressed/head_64.S



```
/*
 * Jump to the relocated address.
 */
	leaq	.Lrelocated(%rbx), %rax
	jmp	*%rax
SYM_CODE_END(startup_64)

#ifdef CONFIG_EFI_STUB
	.org 0x390
SYM_FUNC_START(efi64_stub_entry)
SYM_FUNC_START_ALIAS(efi_stub_entry)
	and	$~0xf, %rsp			/* realign the stack */
	movq	%rdx, %rbx			/* save boot_params pointer */
	call	efi_main
	movq	%rbx,%rsi
	leaq	startup_64(%rax), %rax
	jmp	*%rax
SYM_FUNC_END(efi64_stub_entry)
SYM_FUNC_END_ALIAS(efi_stub_entry)
#endif

	.text
SYM_FUNC_START_LOCAL_NOALIGN(.Lrelocated)
                                                              512,0-1       64%
```

# Linux Boot Process (EFI)

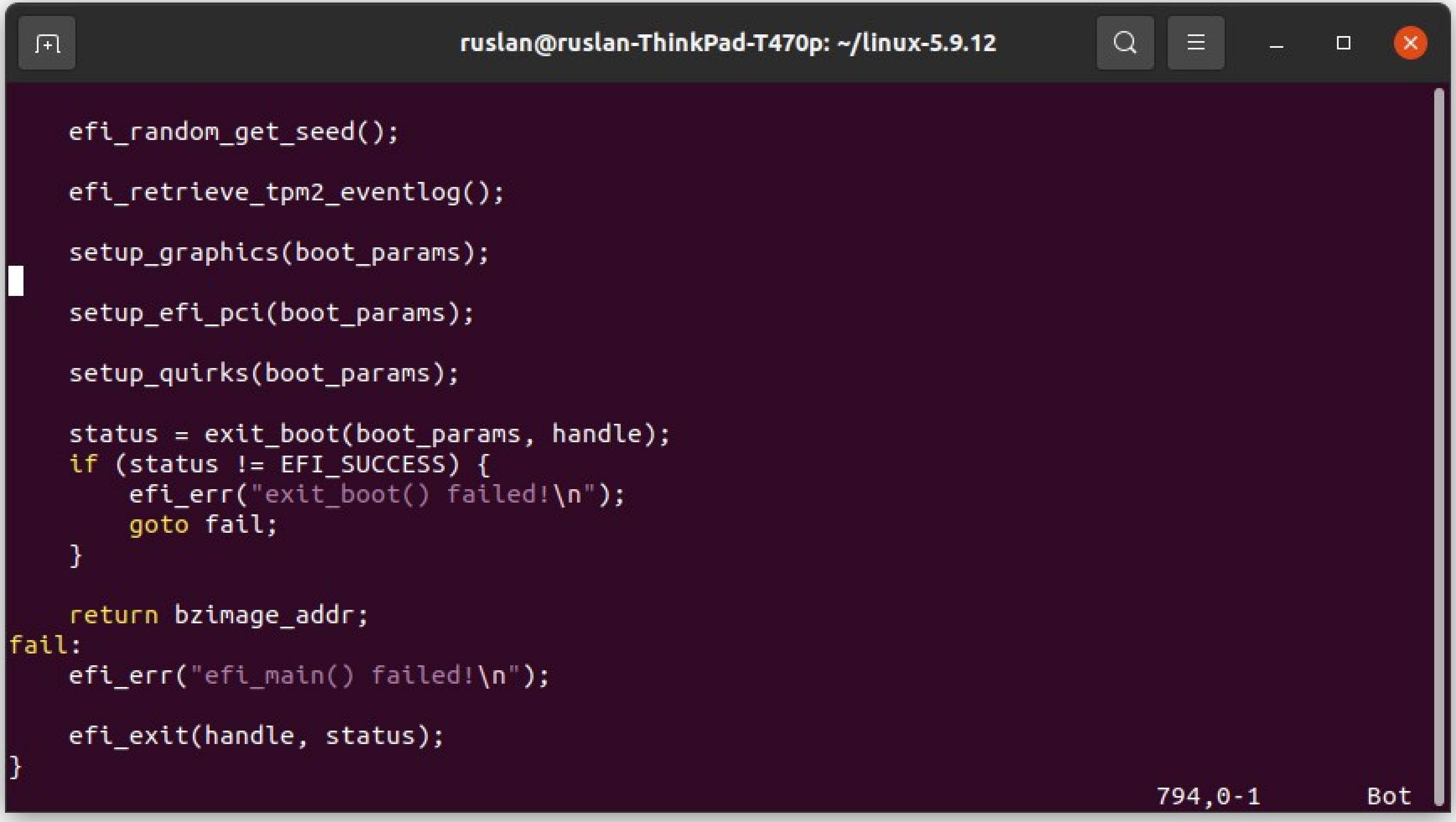
- drivers/firmware/efi/libstub/x86-stub.c



```
/*
 * On success, we return the address of startup_32, which has potentially been
 * relocated by efi_relocate_kernel.
 * On failure, we exit to the firmware via efi_exit instead of returning.
 */
unsigned long efi_main(efi_handle_t handle,
                 efi_system_table_t *sys_table_arg,
                 struct boot_params *boot_params)
{
    unsigned long bzimage_addr = (unsigned long)startup_32;
    unsigned long buffer_start, buffer_end;
    struct setup_header *hdr = &boot_params->hdr;
    efi_status_t status;

    efi_system_table = sys_table_arg;

    /* Check if we were booted by the EFI firmware */
    if (efi_system_table->hdr.signature != EFI_SYSTEM_TABLE_SIGNATURE)
        efi_exit(handle, EFI_INVALID_PARAMETER);

    /*
     * If the kernel isn't already loaded at a suitable address,
                                                                              680,2-5       84%
```

# Linux Boot Process (EFI)

- drivers/firmware/efi/libstub/x86-stub.c



```
    efi_random_get_seed();

    efi_retrieve_tpm2_eventlog();

    setup_graphics(boot_params);

    setup_efi_pci(boot_params);

    setup_quirks(boot_params);

    status = exit_boot(boot_params, handle);
    if (status != EFI_SUCCESS) {
        efi_err("exit_boot() failed!\n");
        goto fail;
    }

    return bzimage_addr;
fail:
    efi_err("efi_main() failed!\n");

    efi_exit(handle, status);
}
                                                                  794,0-1       Bot
```