Gabriele **Proietti Mattia**

# Advanced Operating Systems and Virtualization

## [2] Step 4: Kernel Boot

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v6

# Outline

## [2] Step 4: Kernel Boot

1. **Initial Life of the Linux Kernel**
2. **`startup_32()`**
3. **`start_kernel()`**
    1. A Primer on Memory Organization
    2. Bootmem and Memblock Allocators
    3. Paging Introduction
    4. Paging Initialization
    5. TLB
    6. Final operations and recap

2.1

2. Step 4: Kernel Boot

# Initial Life of the Linux Kernel



DIAG

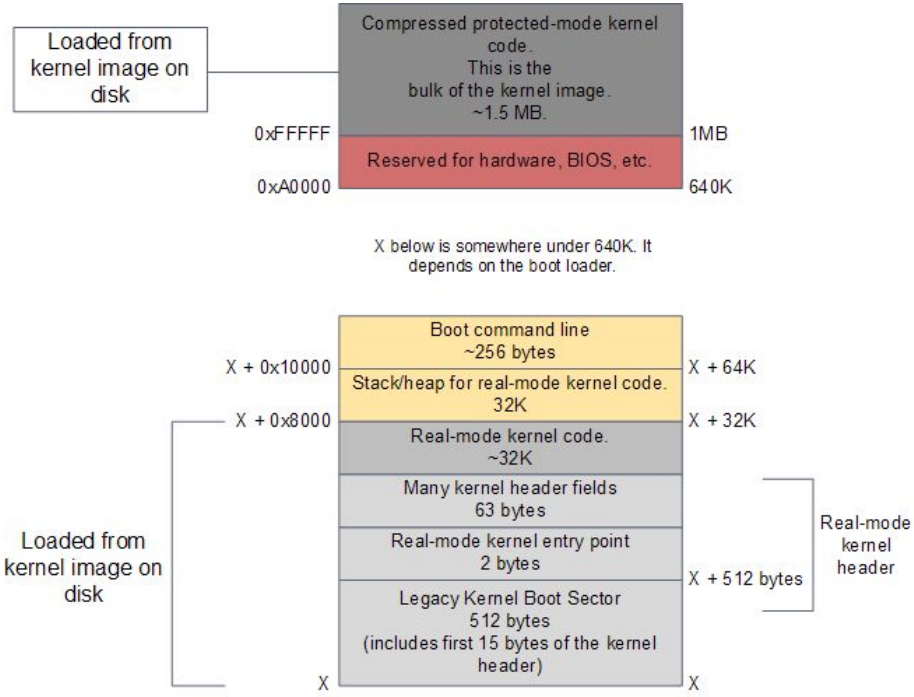
# Outline

1. **BIOS/UEFI**
   Actual hardware setup
2. **Bootloader Stage 1**
   Executes the stage 2 bootloader (skipped for UEFI)
3. **Bootloader Stage 2**
   Loads and starts the kernel
4. **Kernel**
   Takes control and initializes the machine (machine-dependent operations)
5. **Init (or systemd)**
   First process: basic environment initialization
6. **Runlevels/Targets**
   Initializes the user environment

# Initial Life

The Stage 2 Bootloader (or UEFI) loads in RAM the image of the kernel. This image is really different from the one that we have at steady state. We remind that when CPU starts it is in Real Mode with 1MB of addressable memory.

Where is the **entry point**? When the kernel switches to **Protected Mode**?

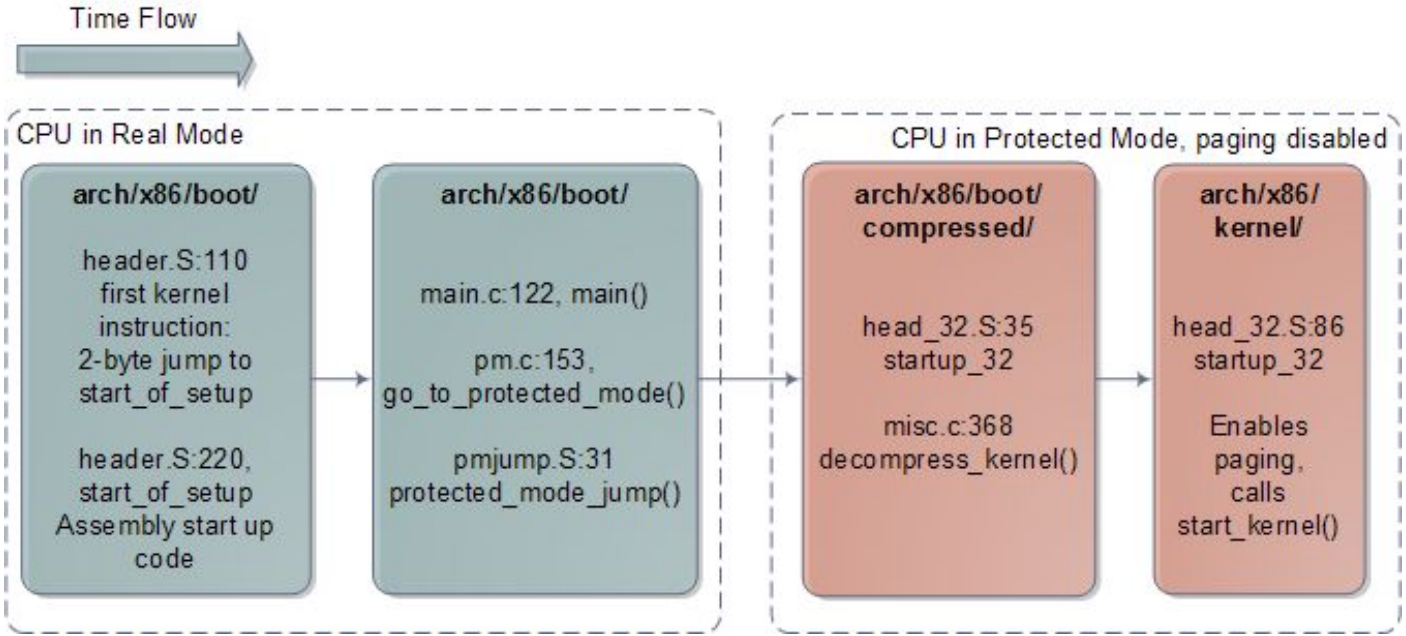



https://manybutfinite.com/post/kernel-boot-process/
http://lxr.linux.no/#linux+v2.6.25.6/Documentation/i386/boot.txt

# From Real to Protected Mode

v2.6

Real Mode




https://manybutfinite.com/post/kernel-boot-process/

# First Instruction

v2.6

Real Mode

The first executed instruction is a 2-byte jump to `start_of_setup` directly written in machine code.

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

https://elixir.bootlin.com/linux/v2.6.25.6/source/arch/x86/boot/header.S#L110

# start_of_setup()

v2.6

Real Mode

This is a short routine that makes some initial setup:

- it sets up a **stack**
- zeroes the **bss** section
- jumps to `main()` in `arch/x86/boot/main.c`

In this portion of code the kernel still runs in real mode, and the function implements part of the **Kernel Boot Protocol**, for example it loads the boot options in memory.

# main()

v2.6

Real Mode

The goal of the `main()` function is to prepare the machine to enter protected and then to the switch. Therefore it:

- enables the A20 line
- sets up the Interrupt Descriptor Table (IDT) and the Global Descriptor Table (GDT)
- sets up memory, asks BIOS which is the available memory for creating a physical address map. As a general rule, the kernel is installed in RAM starting from the physical address 0x00100000, i.e. from the second megabyte. For kernel 2.6, a typical amount of required RAM is 3MB.

In the end the function calls `go_to_protected_mode()` in `arch/x86/boot/pm.c`

# go_to_protected_mode()

V5.11

Real Mode

```
99  /*
100  * Actual invocation sequence
101  */
102 void go_to_protected_mode(void)
103 {
104         /* Hook before leaving real mode, also disables interrupts */
105         realmode_switch_hook();
106
107         /* Enable the A20 gate */
108         if (enable_a20()) {
109                 puts("A20 gate not responding, unable to boot...\n");
110                 die();
111         }
112
113         /* Reset coprocessor (IGNNE#) */
114         reset_coprocessor();
115
116         /* Mask all interrupts in the PIC */
117         mask_all_interrupts();
118
119         /* Actual transition to protected mode... */
120         setup_idt();
121         setup_gdt();
122         protected_mode_jump(boot_params.hdr.code32_start,
123                             (u32)&boot_params + (ds() << 4));
124 }
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/boot/pm.c#L99

# go_to_protected_mode()

V5.11

Real Mode

## Interrupt Descriptor Table

In real mode the Interrupt Vector Table is always at address 0. The IDTR register is set up in the following way:

```
90  /*
91   * Set up the IDT
92   */
93  static void setup_idt(void)
94  {
95          static const struct gdt_ptr null_idt = {0, 0};
96          asm volatile("lidtl %0" : : "m" (null_idt));
97  }
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/boot/pm.c#L93

# go_to_protected_mode()

V5.11

Real Mode

## Global Descriptor Table (GDT)

```
64  static void setup_gdt(void)
65  {
66          /* There are machines which are known to not boot with the GDT
67             being 8-byte unaligned.  Intel recommends 16 byte alignment. */
68          static const u64 boot_gdt[] __attribute__((aligned(16))) = {
69                  /* CS: code, read/execute, 4 GB, base 0 */
70                  [GDT_ENTRY_BOOT_CS] = GDT_ENTRY(0xc09b, 0, 0xfffff),
71                  /* DS: data, read/write, 4 GB, base 0 */
72                  [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0, 0xfffff),
73                  /* TSS: 32-bit tss, 104 bytes, base 4096 */
74                  /* We only have a TSS here to keep Intel VT happy;
75                     we don't actually use it for anything. */
76                  [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096, 103),
77          };
78          /* Xen HVM incorrectly stores a pointer to the gdt_ptr, instead
79             of the gdt_ptr contents.  Thus, make it static so it will
80             stay in memory, at least long enough that we switch to the
81             proper kernel GDT. */
82          static struct gdt_ptr gdt;
83
84          gdt.len = sizeof(boot_gdt)-1;
85          gdt.ptr = (u32)&boot_gdt + (ds() << 4);
86
87          asm volatile("lgdtl %0" : : "m" (gdt));
88  }
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/boot/pm.c#L64

# protected_mode_jump()

V5.11

Real Mode

After setting the initial IDT and GDT, the kernel jumps to protected mode via `protected_mode_jump()` in `arch/x86/boot/pmjump.S`. This routine:

- sets PE in CRo
- issues a ljmp to its very next instruction to load in CS the boot CS sector
- sets up a data segment for flat 32-bit mode
- sets up a temporary stack

```
37          movl    %cr0, %edx
38          orb     $X86_CR0_PE, %dl        # Protected mode
39          movl    %edx, %cr0
40
41          # Transition to 32-bit mode
42          .byte   0x66, 0xea              # ljmpl opcode
43  2:      .long   .Lin_pm32               # offset
44          .word   __BOOT_CS               # segment
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/boot/pmjump.S#L24

# startup_32() #primary

V5.11

Protected

`protected_mode_jump()` jumps into `startup_32()` `arch/x86/boot/compressed/head_32.S` and this routine does the following:

- sets the segments to known values (`__BOOT_DS`)
- loads a new stack
- clears again the BSS section
- determines the actual position in memory via a call/pop (image below)
- calls `extract_kernel()` (previously named `decompress_kernel()`)

```
50  /*
51   * Calculate the delta between where we were compiled to run
52   * at and where we were actually loaded at.  This can only be done
53   * with a short local call on x86.  Nothing  else will tell us what
54   * address we are running at.  The reserved chunk of the real-mode
55   * data at 0x1e4 (defined as a scratch field) are used as the stack
56   * for this calculation. Only 4 bytes are needed.
57   */
58          leal    (BP_scratch+4)(%esi), %esp
59          call    1f
60  1:      popl    %edx
61          addl    $_GLOBAL_OFFSET_TABLE_+(.-1b), %edx
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/boot/compressed/head_32.S#L50

# KASLR

## Kernel Address Space Layout Randomization

In order to prevent that an attacker patches the kernel memory image, at the boot time the kernel **randomly decides** where to decompress itself in memory relying on the most accurate source of entropy available. However, since the kernel in mapped using 2MB aligned pages, the number of valid slots is limited.

*The current layout of the kernel's virtual address space only leaves **512M for the kernel code**—and 1.5G for modules. Since there is no need for that much module space, his patches reduce that to 1G, leaving 1G for the kernel, thus 512 possible slots (as it needs to be 2M aligned). The number of slots may increase when the modules' location is added to KASLR.*

-- https://lwn.net/Articles/569635/

2.2

## 2. Step 4: Kernel Boot

# `startup_32()`



DIAG

# `startup_32()` #secondary

After the decompression the true image of kernel can run, and this is done by a jump to `startup_32()` at `arch/x86/kernel/head_32.S`. This routine sets up the environment for the first Linux process (process `0`):

- initializes the segmentation registers with their final values
- clears again the bss
- builds the page table
- enables paging
- creates the final IDT
- jumps to the architecture-dependent kernel entry point (i.e. `start_kernel()` at `init/kernel.c`)

# Memory

During the initialization the steady-state kernel must take control of the available physical memory. This because it will have to manage it with respect to the virtual address spaces of all processes, in particular it needs to be able to:

- allocate and deallocate memory
- swap

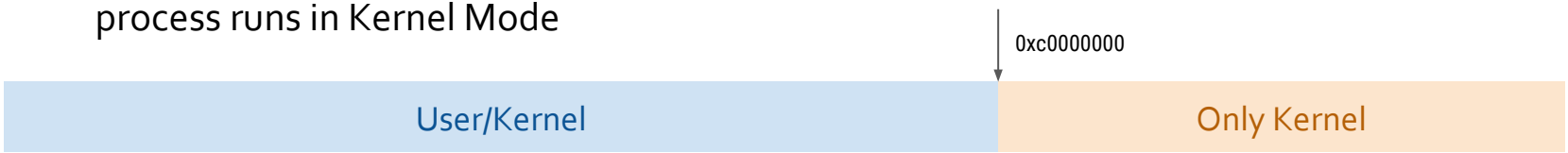
For this reason, upon starting, the kernel must have an early organization setup out of the box. For this reason the kernel use a set of statically generated page tables.

# Process Page Tables

v2.6

On 32bit architecture, the process address space is divided in two parts:

- linear addresses from `0x00000000` to `0xbfffffff` (about 3GB) can be addressed when a process runs in User or Kernel Mode
- linear addresses from `0xc0000000` to `0xffffffff` (about 1GB) can be addressed when a process runs in Kernel Mode




When the process runs in User Mode it issues linear addresses < `0xc0000000`, when in Kernel Mode >= `0xc0000000`.

What should be kept in mind is that addresses lower than `0xc0000000` (value often referred as `PAGE_OFFSET`) depend on the specific process, the others **are the same for every process** and **equal to the corresponding entries of the Master Kernel Page General Directory**.

# Kernel Page Tables

The kernel maintains a set of page tables of its own use rooted at a so-called "**Master Kernel Page Global Directory**". After the system initialization this set of pages tables is never used by any process or kernel thread, but the highest entries will be the reference model for the corresponding entries of the Page Global Directories of every regular process in the system (we will see that every process has a PGD).

The setup of these tables is a two step activity:

1. the kernel first creates a limited address space, including **code**, **data**, the **initial Page Tables** and a **dynamic** area (of 128KB) -- this structure is known at compile time
2. the kernel takes advantage of all of the existing RAM and sets up the page table properly

# Kernel Page Tables

v2.6

## Provisional Kernel Page Tables

A provisional **Page Global Directory** (PGD) is initialized statically during the kernel compilation, while the provisional **Page Tables** are initialized by `startup_32()`.

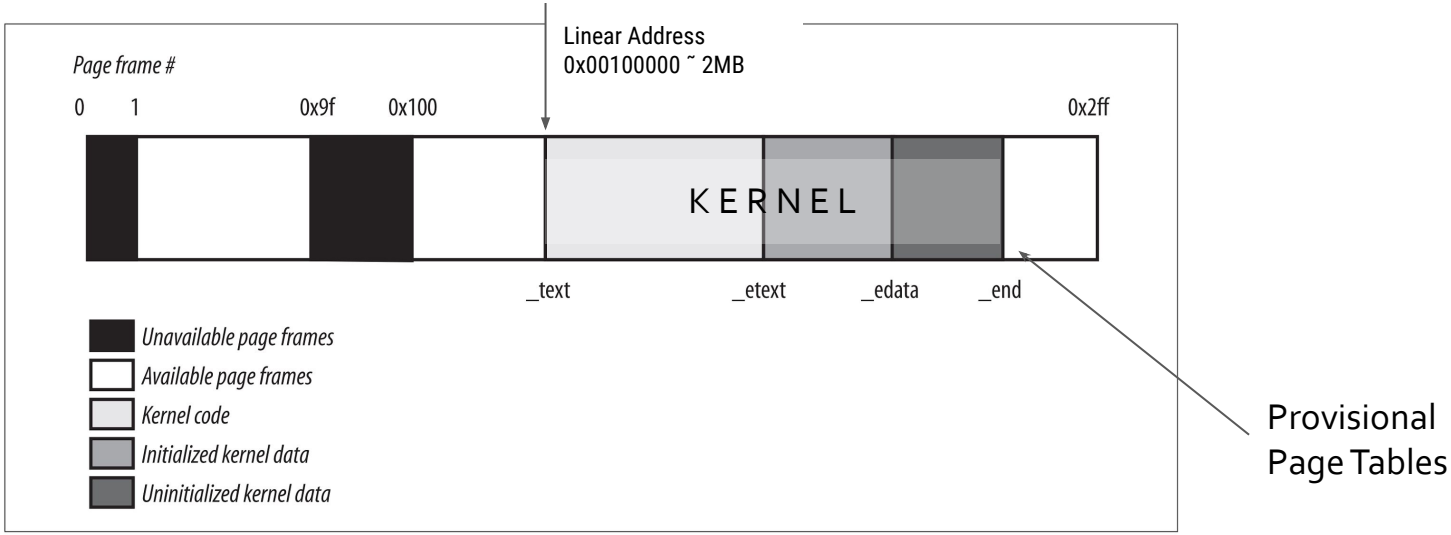



*Figure 2-13. The first 768 page frames (3 MB) in Linux 2.6*

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

# Kernel Page Tables

v2.6

## Provisional Kernel Page Tables

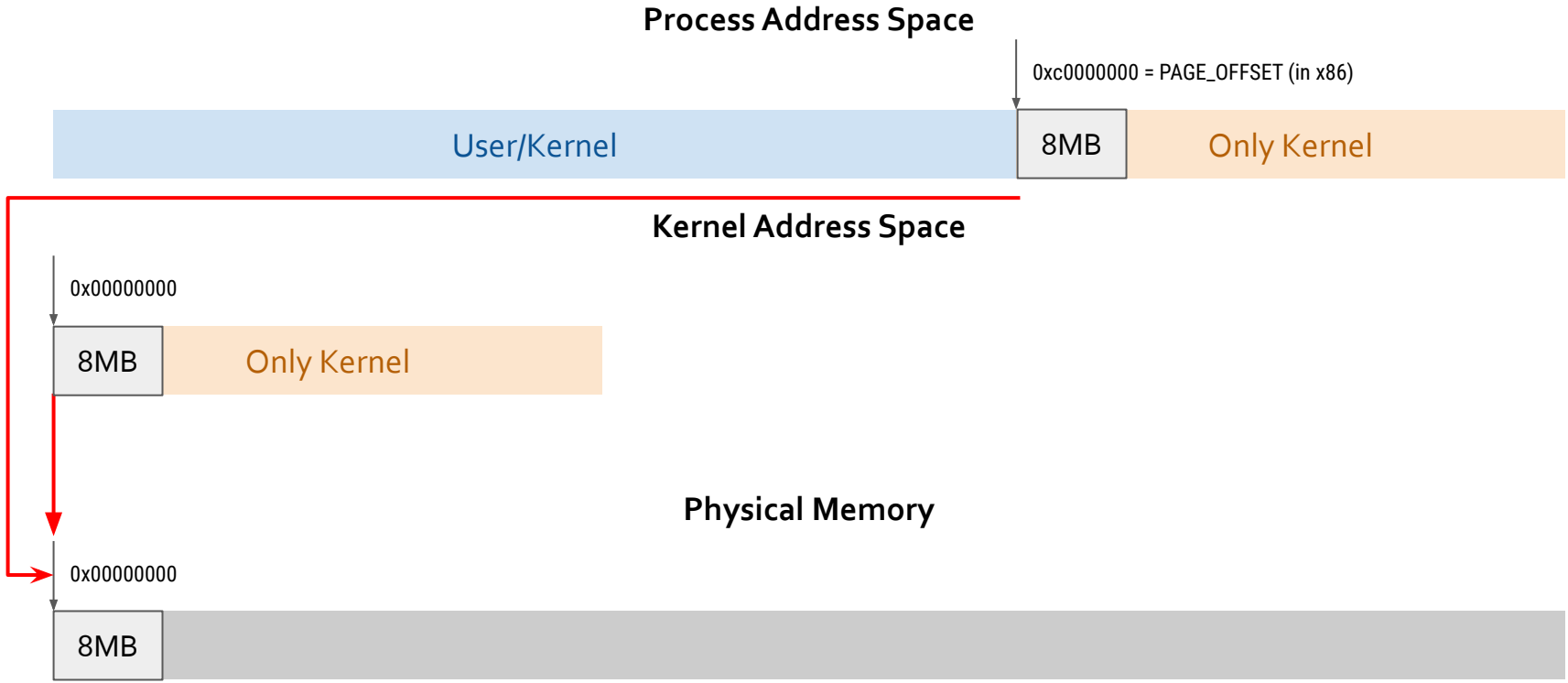
The **Page Global Directory** (PGD) is stored in the `swapper_page_dir` variable. Now suppose that all the kernel segments, the provisional page tables and the dynamic area fits 8MB of RAM. In the early paging, with pages of size 4MB we needed 2 entries in the Page Table.

Now, the objective of this phase of paging is to allow these 8MB of RAM to be easily addressed both in real mode and protected mode. Therefore the kernel must create a mapping from both the linear address `0x00000000` through `0x007fffff` and the linear addresses `0xc0000000` through `0xc07fffff` into the physical `0x00000000` through `0x007fffff`.

This mapping will be explained in Slide 61.

# Kernel Page Tables

## Provisional Kernel Page Tables



# Enabling Paging

V5.11

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/kernel/head_32.S#L31

```
30  /* Physical address */
31  #define pa(X) ((X) - __PAGE_OFFSET)
```

```
493  .globl initial_page_table
494  initial_page_table:
495          .fill 1024,4,0
```

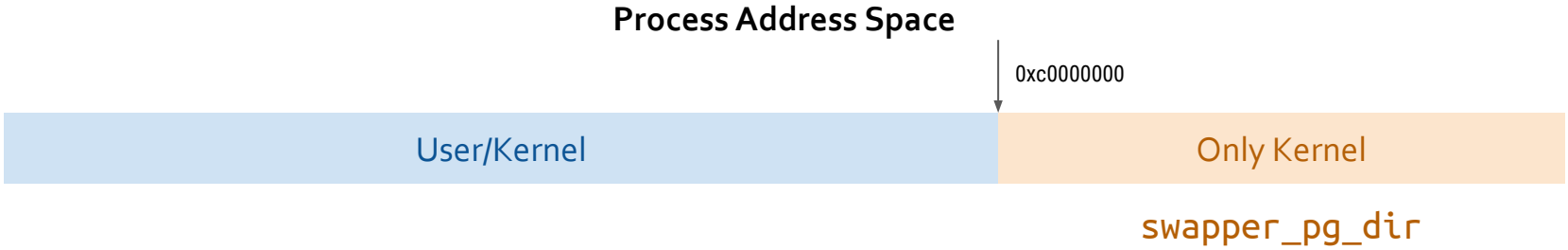
```
250  /*
251   * Enable paging
252   */
253          movl $pa(initial_page_table), %eax
254          movl %eax,%cr3          /* set the page table pointer.. */
255          movl $CR0_STATE,%eax
256          movl %eax,%cr0          /* ..and set paging (PG) bit */
257          ljmp $__BOOT_CS,$1f     /* Clear prefetch and normalize %eip */
258  1:
259          /* Shift the stack pointer to a virtual address */
260          addl $__PAGE_OFFSET, %esp
261
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/kernel/head_32.S#L250

# Kernel-Level MM Data Structures

The main data structures for memory management in the kernel are:

- **Kernel Page Tables,** that keeps the memory mapping for kernel level code and data, it will pointed by `swapper_pg_dir`
- **Core Map,** that keeps the status information for any frame (or page) of the physical memory and the free memory frames for any NUMA node



2.3

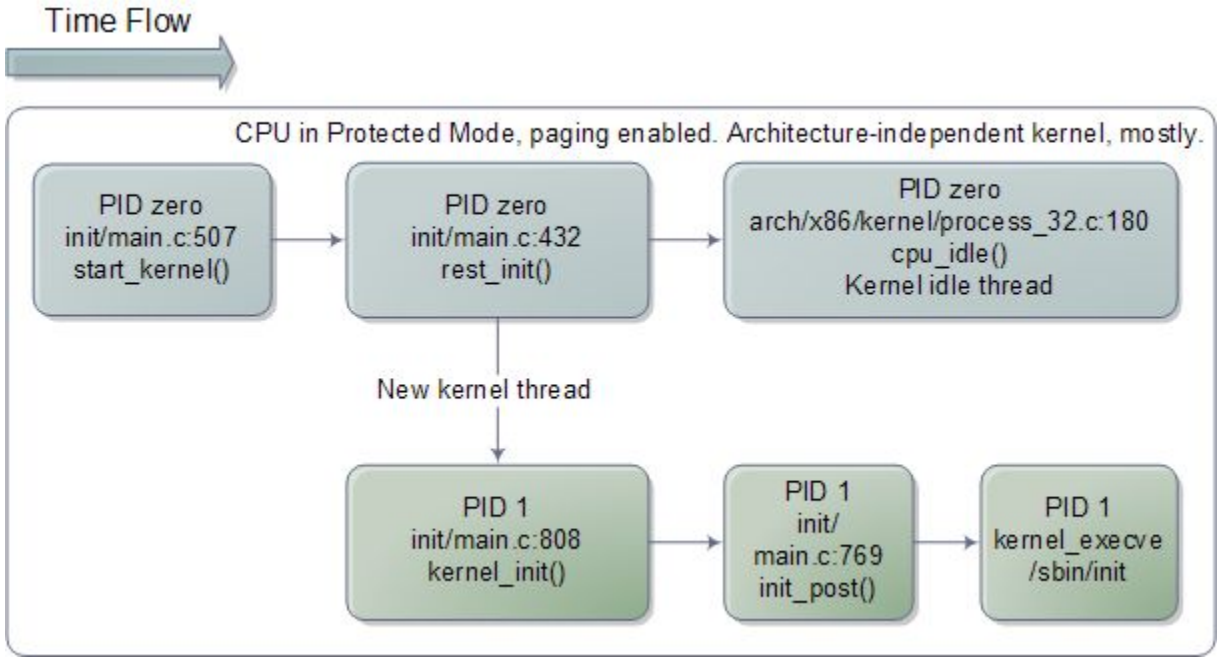
## 2. Step 4: Kernel Boot

# start_kernel()



DIAG

# Kernel Initialization



# Cores Initialization

`start_kernel()` executes on a single core (the master). All of the other cores keep waiting that the master has finished.

The kernel uses the function `smp_processor_id()` for obtaining the ID of the current core. The function is architecture-dependent a written in assembly code by using a specific hardware identification protocol. In modern version it uses the APIC. The function can be used both at kernel startup and at steady state.

# Kernel initialization signature

V5.11

The `start_kernel()` function is declared as

```
asmlinkage __visible void __init __no_sanitize_address start_kernel(void)
```

Where:

- **`asmlinkage`** tells the compiler that the calling convention is such that parameters are passed on stack
- **`__visible`** prevent Link-Time Optimization (since gcc 4.5)
- **`__init`** tells the kernel that the function is only used at initialization phase so memory can be freed' after
- **`__no_sanitize_address`** prevent address sanitizing (since gcc 4.8)

# Main operations

V5.11

The main operations carried out by `start_kernel()` (`init/main.c`) are:

1. `setup_arch()` that initializes the architecture
2. `build_all_zonelists()` - builds the memory zones
3. `page_alloc_init()` / `mem_init()` - the steady state allocator (Buddy System) is initialized and the boot one removed
4. `sched_init()` - initializes the scheduler
5. `trap_init()` - the final IDT is built
6. `time_init()` - the system time is initialized
7. `kmem_cache_init()` - the slab allocator is initialized
8. `arch_call_rest_init()` / `rest_init()` - prepares the environment
   a. `kernel_thread(kernel_init)` - starts the kernel thread for process 1 is created
      i. `kernel_init_freeable()` -> `prepare_namespace()` -> `initrd_load()` - mounts the initramfs, a temporary filesystem used to start the init process
      ii. `run_init_process()` -> `kernel_execve()` - Execute `/bin/init`
   b. `cpu_startup_entry()` -> `do_idle()` - starts the idle process

# setup_arch()

V5.11

The main operations carried out by setup_arch() (/arch/x86/kernel/setup.c) are:

1. load_cr3() - initializes kernel page tables
2. __flush_tlb_all() - flush the TLB
3. init_bootmem() - initializes the bootmem allocator (v < 5)
4. e820__memory_setup() / e820__reserve_resources() - initializes the available memory (also for memblock allocator)
5. x86_init.paging.pagetable_init() -> native_pagetable_init() -> paging_init() - initializes paging

2.3.1

2. Step 4: Kernel Boot
3. start_kernel()

# A Primer on Memory Organization



DIAG

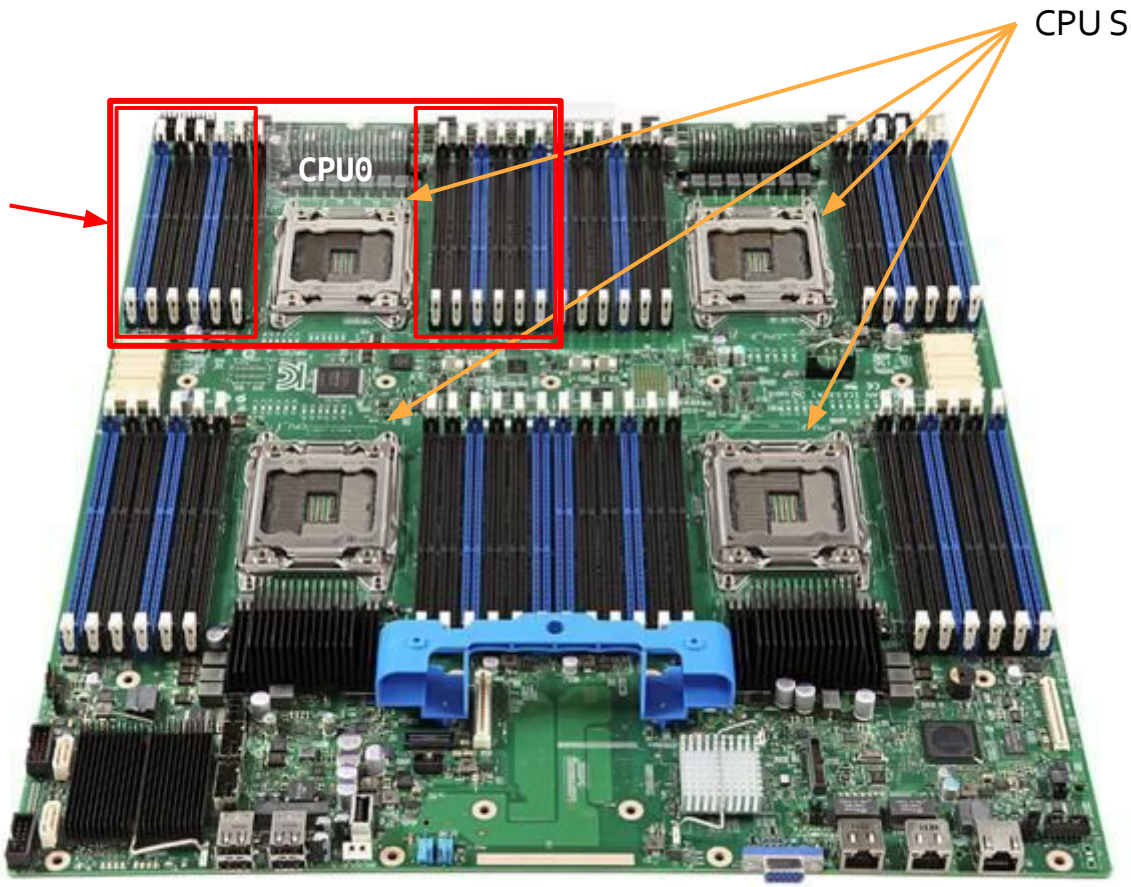
# NUMA

Linux is available for a great number of architectures, for this reason a machine independent way of describing memory is needed.

Large scale machines memory may be arranged into banks that incur a different access delay depending on the distance from the CPU. For this reason a memory bank can be assigned to each CPU, or a bank can be suitable for Direct Memory Access (DMA) near the devices.

Each of these banks is called in linux a **node** and the concept of accessing the memory in nodes is called *Non-Uniform Memory Access* (**NUMA**) (with a single node we have a UMA architecture). Each node is represented by the `struct pg_data_t` and all nodes are kept in linked list.

# NUMA



# Zones

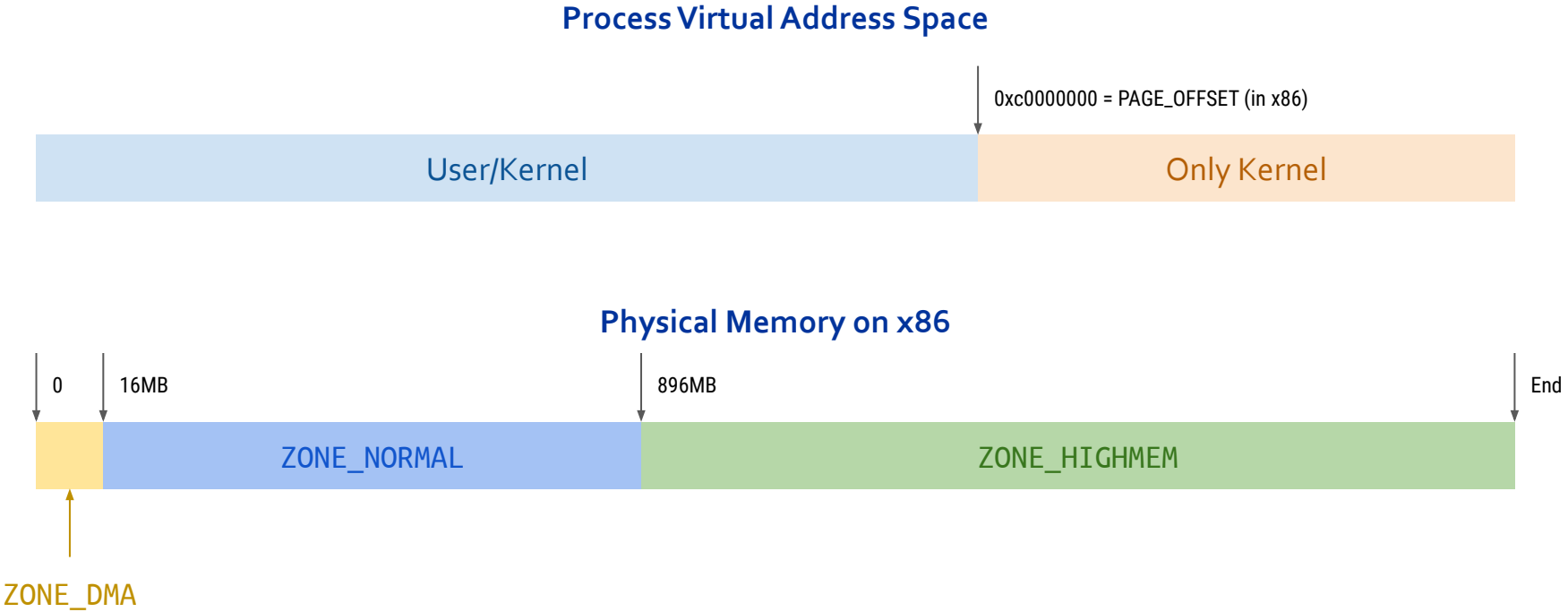
Each node is divided in a number of blocks called zones, which represents ranges within the memory. On x86 there are three kinds of zone:

- `ZONE_DMA` is **directly mapped** by the kernel in the lower part of memory and it is destined to ISA (Industry Standard Architecture) devices, in x86 first 16 MB
- `ZONE_NORMAL` is **directly mapped** by the kernel into the upper region of the linear address space, in x86 from 16MB to 896MB
- `ZONE_HIGHMEM` is the remaining available memory and it is **not directly mapped** by the kernel, in x86 from 896MB to end of memory.

The Page table is usually located at the top beginning of `ZONE_NORMAL`. To access memory between 1GB and 4GB the kernel temporarily maps pages from high memory to `ZONE_NORMAL`.

`ZONE_NORMAL` is fixed in size, addressing 16GiB can require 176MB of data structures!

# Zones


Process Virtual Address Space
0xc0000000 = PAGE_OFFSET (in x86)
User/Kernel
Only Kernel
Physical Memory on x86
0
16MB
896MB
End
ZONE_NORMAL
ZONE_HIGHMEM
ZONE_DMA

2.3.2

2. Step 4: Kernel Boot
3. start_kernel()

# Bootmem and Memblock Allocators



DIAG

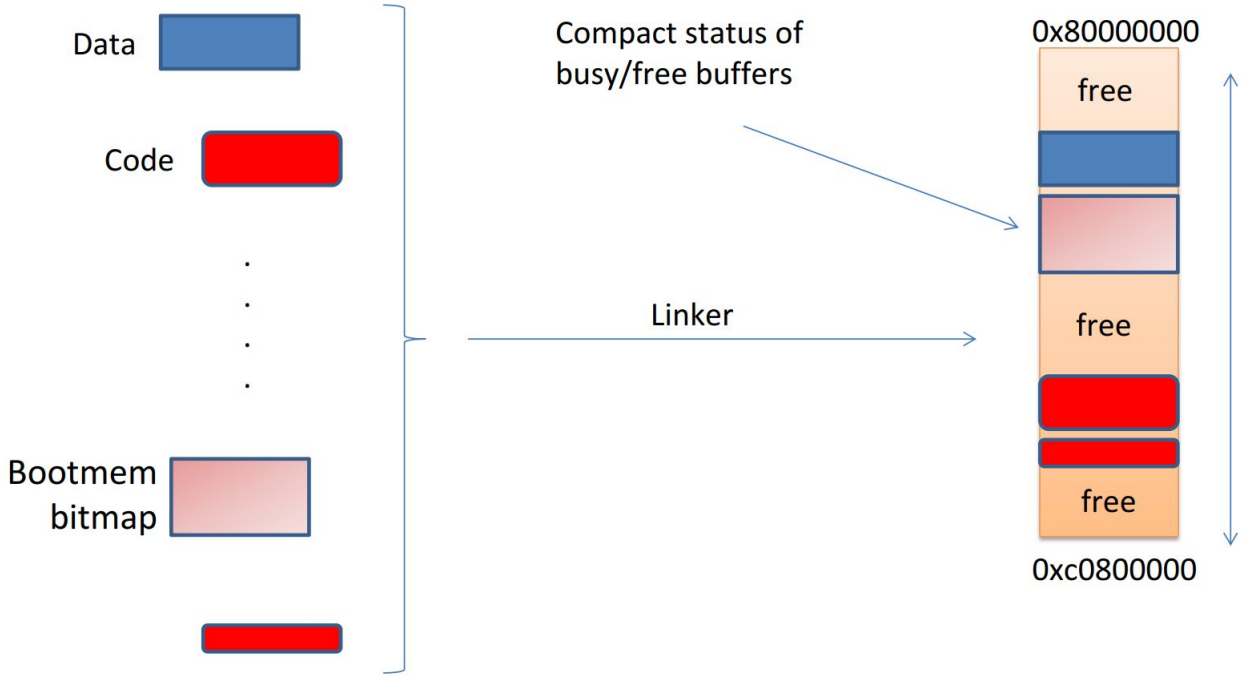
# Bootmem allocator

In the previous section we concluded that the memory map of the initial kernel image is known at **compile** time. It's impractical to initialize all of the core kernel memory structures at compile time. The Linux kernel have a link-time memory manager, embedded into the kernel image, called **bootmem allocator** (linux/bootmem.h).

The Bootmem allocator relies on **bitmaps** (instead of linked list of free blocks) that tells if any 4KB page in the currently reachable memory is busy or free. It also offers **API** (only at boot time) to get free buffers, i.e. sets of contiguous page-aligned areas.

Bootmem allocator is a *First Fit* allocator. To satisfy the allocations that are less than a page, the allocator records the last allocated Page Frame Number (PFN) and the offset. Subsequent allocations are stored in the same page.

# Bootmem organization



# Initializing bootmem

The information used by the bootmem allocator is represented by `struct bootmem_data`. An array to hold up to `MAX_NUMNODES` such structures is statically allocated and then it is discarded when the system initialization completes. Each entry in this array corresponds to a node with memory. For UMA systems only entry o is used.

# Initializing bootmem

The bootmem allocator is initialized during early architecture specific setup. Each architecture is required to supply a `setup_arch()` (called by start_kernel()) function which, among other tasks, is responsible for acquiring the necessary parameters to initialise the boot memory allocator. These parameters define limits of usable physical memory:

- **`min_low_pfn`** - the lowest PFN that is available in the system
- **`max_low_pfn`** - the highest PFN that may be addressed by low memory (ZONE_NORMAL)
- **`max_pfn`** - the last PFN available to the system.

After those limits are determined, the `init_bootmem()` or `init_bootmem_node()` function should be called to initialize the bootmem allocator. The UMA case should use the `init_bootmem` function. It will initialize `contig_page_data` structure that represents the only memory node in the system. In the NUMA case the `init_bootmem_node` function should be called to initialize the bootmem allocator for each node.

# Bootmem APIs

```
#include <linux/bootmem.h>
```

```
void *alloc_bootmem(unsigned long size);
```

Allocate size number of bytes from ZONE_NORMAL. The allocation will be aligned to the L1 hardware cache to get the maximum benefit from the hardware cache

```
void *alloc_bootmem_low(unsigned long size);
```

Allocate size number of bytes from ZONE_DMA. The allocation will be aligned to the L1 hardware cache

```
void *alloc_bootmem_pages(unsigned long size);
```

Allocate size number of bytes from ZONE_NORMAL aligned on a page size so that full pages will be returned to the caller

```
void *alloc_bootmem_low_pages(unsigned long size);
```

Allocate size number of bytes from ZONE_NORMAL aligned on a page size so that full pages will be returned to the caller

```
void free_bootmem(unsigned long addr, unsigned long size);
```

Bootmem API is only available for code linked in the kernel image.

https://www.kernel.org/doc/gorman/html/understand/understandoo8.html

# Memblock allocator

V5.11

**Memblock** is a method of managing memory regions during the early boot period when the usual kernel memory allocators are not up and running. The memblock allocator, differently from the bootmem does not use bitmaps for keeping track of allocated regions, but collections of regions.

Memblock views the system memory as collections of contiguous regions. There are several types of these collections:

- `memory` - describes the physical memory available to the kernel; this may differ from the actual physical memory installed in the system, for instance when the memory is restricted with mem= command line parameter
- `reserved` - describes the regions that were allocated
- `physmem` - describes the actual physical memory available during boot regardless of the possible restrictions and memory hot(un)plug; the physmem type is only available on some architectures.

https://www.kernel.org/doc/html/latest/core-api/boot-time-mm.html

# Memblock Allocator

V5.11

## Data Structures

```c
/**
 * struct memblock - memblock allocator metadata
 * @bottom_up: is bottom up direction?
 * @current_limit: physical address of the current allocation limit
 * @memory: usable memory regions
 * @reserved: reserved memory regions
 */
struct memblock {
        bool bottom_up;  /* is bottom up direction? */
        phys_addr_t current_limit;
        struct memblock_type memory;
        struct memblock_type reserved;
};
```

https://elixir.bootlin.com/linux/v5.11.2/source/include/linux/memblock.h#L74

```c
/**
 * struct memblock_region - represents a memory region
 * @base: base address of the region
 * @size: size of the region
 * @flags: memory region attributes
 * @nid: NUMA node id
 */
struct memblock_region {
        phys_addr_t base;
        phys_addr_t size;
        enum memblock_flags flags;
#ifdef CONFIG_NEED_MULTIPLE_NODES
        int nid;
#endif
};
```

https://elixir.bootlin.com/linux/v5.11.2/source/include/linux/memblock.h#L42

```c
/**
 * struct memblock_type - collection of memory regions of certain type
 * @cnt: number of regions
 * @max: size of the allocated array
 * @total_size: size of all regions
 * @regions: array of regions
 * @name: the memory type symbolic name
 */
struct memblock_type {
        unsigned long cnt;
        unsigned long max;
        phys_addr_t total_size;
        struct memblock_region *regions;
        char *name;
};
```

https://elixir.bootlin.com/linux/v5.11.2/source/include/linux/memblock.h#L58

# Memblock allocator

## Architecture

Each region is represented by struct `memblock_region` that defines the region extents, its attributes and NUMA node id on NUMA systems. Every memory type is described by the struct `memblock_type` which contains an array of memory regions along with the allocator metadata.

The "`memory`" and "`reserved`" types are nicely wrapped with `struct memblock`. This structure is statically initialized at build time. The region arrays are initially sized to `INIT_MEMBLOCK_REGIONS` for "memory" and `INIT_MEMBLOCK_RESERVED_REGIONS` for "reserved". The region array for "physmem" is initially sized to `INIT_PHYSMEM_REGIONS`.

The `memblock_allow_resize()` enables automatic resizing of the region arrays during addition of new regions. This feature should be used with care so that memory allocated for the region array will not overlap with areas that should be reserved, for example initrd.

# Memblock allocator

## API

After the initialization of `memory` regions done by setup_arch() with functions `memblock_add()` or `memblock_add_node()` functions. We can use the following APIs:

- `memblock_phys_alloc*()` - these functions return the **physical** address of the allocated memory
  - **`memblock_phys_alloc_range`**`(phys_addr_t size, phys_addr_t align, phys_addr_t start, phys_addr_t end);`
  - **`memblock_phys_alloc_try_nid`**`(phys_addr_t size, phys_addr_t align, int nid);`
  - **`memblock_phys_alloc`**`(phys_addr_t size, phys_addr_t align)`
- `memblock_alloc*()` - these functions return the **virtual** address of the allocated memory:
  - **`memblock_alloc`**`(phys_addr_t size, phys_addr_t align)`
  - **`memblock_alloc_raw`**`(phys_addr_t size, phys_addr_t align)`
  - **`memblock_alloc_from`**`(phys_addr_t size, phys_addr_t align, phys_addr_t min_addr)`
  - **`memblock_alloc_low`**`(phys_addr_t size, phys_addr_t align)`
  - **`memblock_alloc_node`**`(phys_addr_t size, phys_addr_t align, int nid)`

# Memblock allocator

## After boot

As the system boot progresses, the architecture specific `mem_init()` function frees all the memory to the buddy page allocator.

Unless an architecture enables `CONFIG_ARCH_KEEP_MEMBLOCK`, the memblock data structures (except "physmem") will be discarded after the system initialization completes.

# Memblock use case

## e820

In the x86 architecture, the e820 is the shorthand for obtaining the memory map of the system. At boot time, by looking at the `dmesg` you will find:

```
[   +0.000000] e820: BIOS-provided physical RAM map:
[   +0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009f7ff] usable
[   +0.000000] BIOS-e820: [mem 0x000000000009f800-0x000000000009ffff] reserved
[   +0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[   +0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000bfd9ffff] usable
[   +0.000000] BIOS-e820: [mem 0x00000000bfda0000-0x00000000bfdd0fff] ACPI NVS
[   +0.000000] BIOS-e820: [mem 0x00000000bfdd1000-0x00000000bfdfffff] ACPI data
[   +0.000000] BIOS-e820: [mem 0x00000000bfe00000-0x00000000bfefffff] reserved
[   +0.000000] BIOS-e820: [mem 0x00000000e0000000-0x00000000efffffff] reserved
[   +0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000ffffffff] reserved
[   +0.000000] BIOS-e820: [mem 0x0000000100000000-0x000000043effffff] usable
```

The e820 subsystem must mark some specific areas to be reserved, for example the can be reserved for the kernel, for BIOS facilities or drivers. The code on the right just does this by using memblock.

V5.11

```
1129  /*
1130   * Mark E820 reserved areas as busy for the resource manager;
1131   */
1132
1133  static struct resource __initdata *e820_res;
1134
1135  void __init e820__reserve_resources(void)
1136  {
1137          int i;
1138          struct resource *res;
1139          u64 end;
1140
1141          res = memblock_alloc(sizeof(*res) * e820_table->nr_entries,
1142                               SMP_CACHE_BYTES);
1143          if (!res)
1144                  panic("%s: Failed to allocate %zu bytes\n", __func__,
1145                        sizeof(*res) * e820_table->nr_entries);
1146          e820_res = res;
1147
1148          for (i = 0; i < e820_table->nr_entries; i++) {
1149                  struct e820_entry *entry = e820_table->entries + i;
1150
1151                  end = entry->addr + entry->size - 1;
1152                  if (end != (resource_size_t)end) {
1153                          res++;
1154                          continue;
1155                  }
1156                  res->start = entry->addr;
1157                  res->end   = end;
1158                  res->name  = e820_type_to_string(entry);
1159                  res->flags = e820_type_to_iomem_type(entry);
1160                  res->desc  = e820_type_to_iores_desc(entry);
1161
1162                  /*
1163                   * Don't register the region that could be conflicted with
1164                   * PCI device BAR resources and insert them later in
1165                   * pcibios_resource_survey():
1166                   */
1167                  if (do_mark_busy(entry->type, res)) {
1168                          res->flags |= IORESOURCE_BUSY;
1169                          insert_resource(&iomem_resource, res);
1170                  }
1171                  res++;
1172          }
```

https://elixir.bootlin.com/linux/v5.11.2/source/arch/x86/kernel/e820.c#L1129

# Superseding

In recent versions of the kernel (5+), the bootmem allocator has been removed in favour of the memblock allocator on almost all architectures. See this patch https://lwn.net/Articles/764807/

## mm: remove bootmem allocator

**From**: Mike Rapoport <rppt-AT-linux.vnet.ibm.com>
**To**: linux-mm-AT-kvack.org
**Subject**: [PATCH 00/30] mm: remove bootmem allocator
**Date**: Fri, 14 Sep 2018 15:10:15 +0300
**Message-ID**: <1536927045-23536-1-git-send-email-rppt@linux.vnet.ibm.com>

Hi,

These patches switch early memory management to use memblock directly without any bootmem compatibility wrappers. As the result both bootmem and nobootmem are removed.

The patchset survived allyesconfig builds on arm, arm64, i386, mips, nds32, parisc, powerpc, riscv, s390 and x86 and most of the *_defconfig builds for all architectures except unicore32.

The patchset is based on v4.19-rc3-mmotm-2018-09-12-16-40, so I needed a small PSI fix from [1] for some of the builds.

I did my best to verify that the failures are not caused by my changes, but I may have missed something. Most defconfig build failures I've seen were caused by assembler being unhappy about unsupported opcode, wrong encoding

2.3.3

2. Step 4: Kernel Boot
3. start_kernel()

# Paging Introduction



DIAG

# Pages handling

Prior to version 2.6.11 the Linux paging model consisted of 3 indirection levels, next versions introduced another level of indirections for a total of 4.

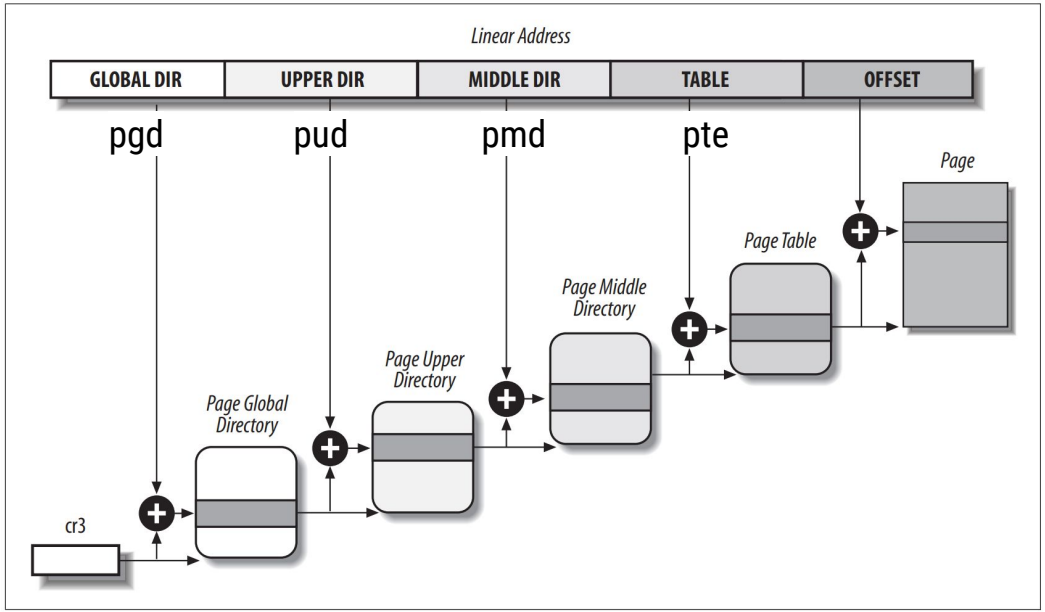



*Figure 2-12. The Linux paging model*

# Splitting the address

For splitting the linear address there are three kinds of macros that can be used:

- `SHIFT` macros specify the length in bits mapped to each PT level
- `MASK` macros AND'd with an address mask out all the upper bits and they are often used for understanding if an address is aligned to a given level within the page table
- `SIZE` macros reveal how many bytes are addressed by each entry at each level

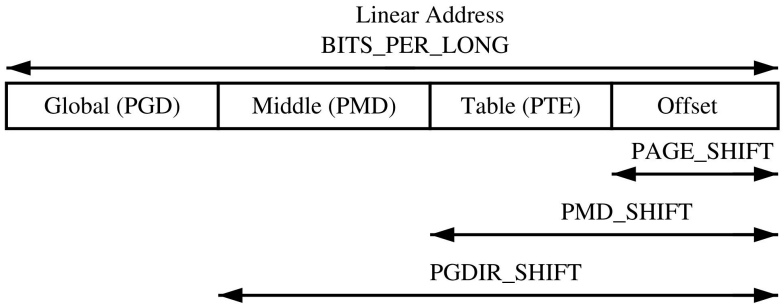



**Figure 3.2.** Linear Address Bit Size Macros

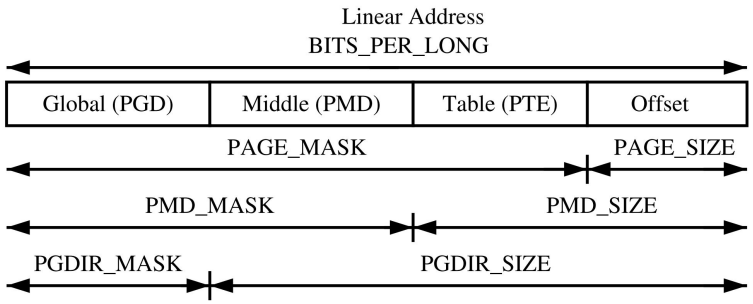



**Figure 3.3.** Linear Address Size and Mask Macros

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

# Configuring the PT

Those macros are declared in the following source files:

- arch/x86/include/asm/pgtable-2level_types.h
- arch/x86/include/asm/pgtable-3level_types.h
- arch/x86/include/asm/pgtable_64_types.h

With other kinds of macros like PTRS_PER_* which describe the number of entries in PTs

```
27  /*
28   * PGDIR_SHIFT determines what a top-level page table entry can map
29   */
30  #define PGDIR_SHIFT     30
31  #define PTRS_PER_PGD    4
32
33  /*
34   * PMD_SHIFT determines the size of the area a middle-level
35   * page table can map
36   */
37  #define PMD_SHIFT       21
38  #define PTRS_PER_PMD    512
39
40  /*
41   * entries per page directory level
42   */
43  #define PTRS_PER_PTE    512
```

https://elixir.bootlin.com/linux/v5.11.2/source/arch/x86/include/asm/pgtable-3level_types.h#L27

# Page Table data structures

As already introduced earlier, swapper_pg_dir keeps the virtual memory address of the PGD (PDE) portion of the kernel page table. The data structure is initialized at compile time, depending on the memory layout defined for the kernel bootable image.

Any entry in the PGD is accessed with displacement, but the main types for defining page table entries are explicitly defined, even if they are just unsigned integers:

```
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
```

This is done essentially for enforcing type protection and for supporting PAE (where additional 4 bits are used for addressing more than 4GB of RAM).

An additional structure is used for storing page protection bits: `pgprot_t`.

# (C and weak typing)

Remember that C language is weak typed, therefore the following code will compile and execute with nor error or warning:

```
typedef unsigned long pgd_t;
typedef unsigned long pte_t;
pgd_t x; pte_t y;
x = y;
y = x;
```

# Bit fields

V5.11

```
12  #define _PAGE_BIT_PRESENT       0       /* is present */
13  #define _PAGE_BIT_RW            1       /* writeable */
14  #define _PAGE_BIT_USER          2       /* userspace addressable */
15  #define _PAGE_BIT_PWT           3       /* page write through */
16  #define _PAGE_BIT_PCD           4       /* page cache disabled */
17  #define _PAGE_BIT_ACCESSED      5       /* was accessed (raised by CPU) */
18  #define _PAGE_BIT_DIRTY         6       /* was written to (raised by CPU) */
19  #define _PAGE_BIT_PSE           7       /* 4 MB (or 2MB) page */
20  #define _PAGE_BIT_PAT           7       /* on 4KB pages */
21  #define _PAGE_BIT_GLOBAL        8       /* Global TLB entry PPro+ */
22  #define _PAGE_BIT_SOFTW1        9       /* available for programmer */
23  #define _PAGE_BIT_SOFTW2        10      /* " */
24  #define _PAGE_BIT_SOFTW3        11      /* " */
25  #define _PAGE_BIT_PAT_LARGE     12      /* On 2MB or 1GB pages */
26  #define _PAGE_BIT_SOFTW4        58      /* available for programmer */
27  #define _PAGE_BIT_PKEY_BIT0     59      /* Protection Keys, bit 1/4 */
28  #define _PAGE_BIT_PKEY_BIT1     60      /* Protection Keys, bit 2/4 */
29  #define _PAGE_BIT_PKEY_BIT2     61      /* Protection Keys, bit 3/4 */
30  #define _PAGE_BIT_PKEY_BIT3     62      /* Protection Keys, bit 4/4 */
31  #define _PAGE_BIT_NX            63      /* No execute: only valid after cpuid check */
```

https://elixir.bootlin.com/linux/v5.11.2/source/arch/x86/include/asm/pgtable_types.h#L12

# Bit fields and Masks

V5.11

Type casting macros are defined in asm/page.h which takes the previous types and returns the relevant part of the struct. They are `pte_val()`, `pmd_val()`, `pgd_val()` and `pgprot_val()`. For reverse type casting we have `__pte()`, `__pmd()`, `__pgd()`, `__pgprot()`.

In the following example code, a check if a page is present is carried out:

```
93  /*
94   * Create a page table and place a pointer to it in a middle page
95   * directory entry:
96   */
97  static pte_t * __init one_page_table_init(pmd_t *pmd)
98  {
99          if (!(pmd_val(*pmd) & _PAGE_PRESENT)) {
100                 pte_t *page_table = (pte_t *)alloc_low_page();
101
102                 paravirt_alloc_pte(&init_mm, __pa(page_table) >> PAGE_SHIFT);
103                 set_pmd(pmd, __pmd(__pa(page_table) | _PAGE_TABLE));
104                 BUG_ON(page_table != pte_offset_kernel(pmd, 0));
105         }
106
107         return pte_offset_kernel(pmd, 0);
108 }
```

# Different PD Entries

Different kind of page entries are again described with macros in `/arch/x86/include/asm/pgtable_types.h`

```
#define _PAGE_TABLE \
(_PAGE_PRESENT | _PAGE_RW | \
_PAGE_USER | _PAGE_ACCESSED | \
_PAGE_DIRTY)

#define _KERNPG_TABLE \
(_PAGE_PRESENT | _PAGE_RW | \
_PAGE_ACCESSED | _PAGE_DIRTY)
```

2.3.4

2. Step 4: Kernel Boot
3. start_kernel()

# Paging Initialization

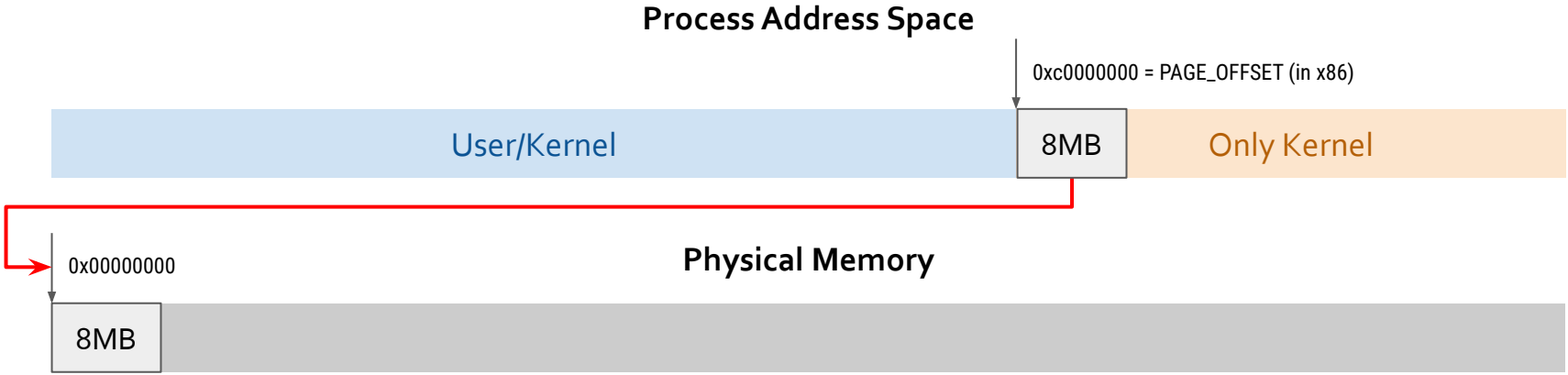


DIAG

# Bootstrapping

v2.6

As already introduced, `startup_32()` enables the Paging unit. While the kernel code is compiled with base address at `PAGE_OFFSET + 1MB`, the kernel is actually loaded at the beginning of physical memory. The initialization of kernel page tables begins at compile time, statically defining an array called `swapper_pg_dir` (at `0x00101000`), that establishes page table entries for **2 pages of 4MB each**, `pg0` and `pg1`. These two pointers covers the addresses from 1MB to 9MB but they are placed at PAGE_OFFSET + 1MB.

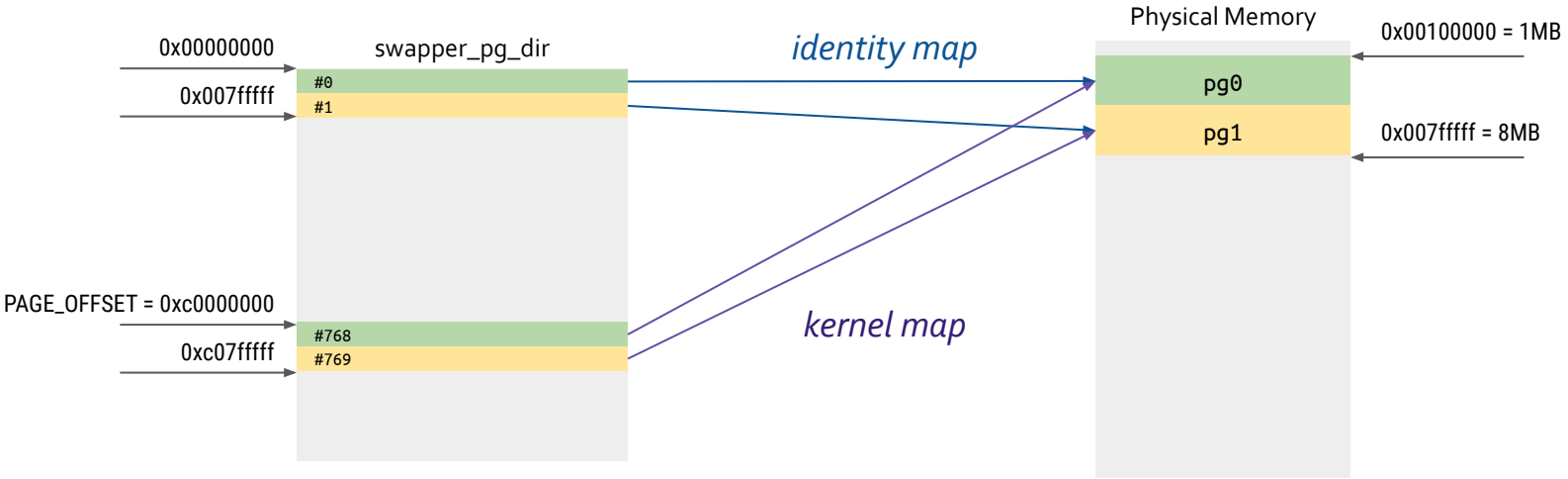


# Bootstrapping

The 8MB of addressable memory must be addressed both in real mode than in protected mode. For this reason, for that memory area the physical address must be equal to the virtual one. This strategy is realized by declaring statically **4 entries** in the `swapper_pg_dir`:

- Entry `0` and `0x300` (768) point to `pg0`
- Entry 1 and `0x301` (769) point to `pg1`

These entries have set bits P,R/W,U/S and cleared A,D,PCD,PWD and Page Size.



# Bootstrapping

V2.4

The **Provisional Page Table** with only two pages is set with the following assembly instructions

```
movl $swapper_pg_dir-0xc0000000,%eax
movl %eax,%cr3          /* set the page table pointer.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0          /* ..and set paging (PG) bit */
```

# Bootstrapping

V2.4

The rest of kernel page tables are initialized by `paging_init()` called by `setup_arch()`.

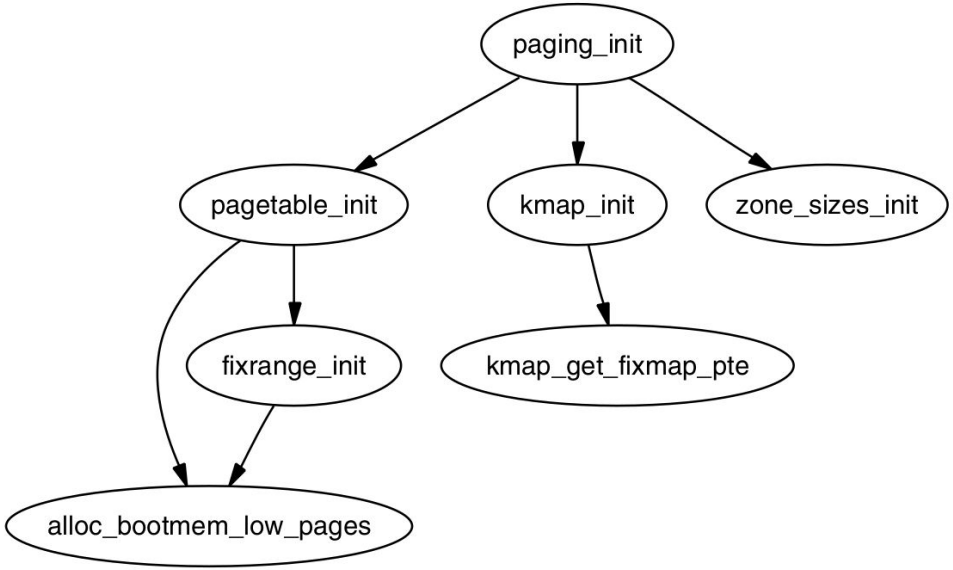



**Figure 3.4.** Call Graph: `paging_init()`

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

# pagetable_init()

V2.4

The initialization of kernel page tables starts with function `paging_init()` that initializes the necessary pages for addressing ZONE_DMA and ZONE_NORMAL from PAGE_OFFSET.

```
for (; i < PTRS_PER_PGD; pgd++, i++) {
    vaddr = i*PGDIR_SIZE; /* i is set to map from 3 GB */
    if (end && (vaddr >= end)) break;
    pmd = (pmd_t *) pgd;/* pgd initialized to (swapper_pg_dir+i) */
    .........
    for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
        .........
        pte_base = pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
            vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
            if (end && (vaddr >= end)) break;
            .........
            *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
        }
        set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte_base)));
        .........
    }
}
```

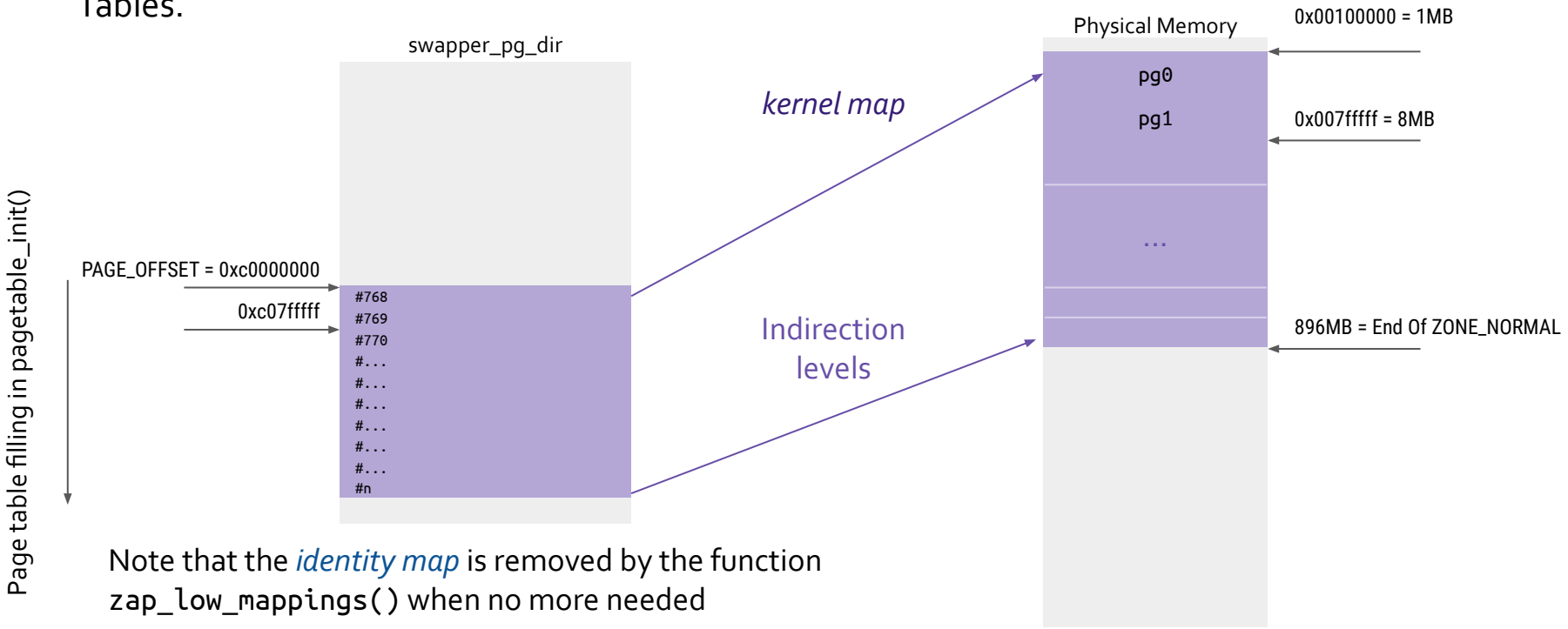
The starting address `vaddr` is set to start from 3GB because we are mapping the virtual addresses of kernel pages.

The kernel prefers to use 4MB pages if the CPU support them, this for reducing the TLB miss rate and speeding up the address translation.

https://elixir.bootlin.com/linux/2.4.31/source/arch/i386/mm/init.c#L205

# Bootstrapping

After pagetable_init() execution, all the ZONE_NORMAL and ZONE_DMA is directly **mapped** in the kernel. Memory is not allocated, is just mapped. Memory is allocated only for the Page Tables.




Note that the *identity map* is removed by the function `zap_low_mappings()` when no more needed

# set_pmd() and __pa()/__va()

V2.4

```
#define set_pmd(pmdptr, pmdval) (*(pmdptr) = pmdval)
```

Parameters are:

- `pmdptr`, pointing to an entry of the PMD, of type pmd_t. The value to assign, of pmd_t type is computed by using the macro

```
#define __pa(x)((unsigned long)(x)-PAGE_OFFSET)
```

Linux sets up a direct mapping from the physical address `0` to the virtual address `PAGE_OFFSET` at 3GB on x86. The opposite can be done using the `__va(x)` macro.

# mk_pte_sys()

V2.4

The function creates a page table entry given the physical address and the protection metadata.

```
311  /* This takes a physical page address that is used by the remapping functions */
312  #define mk_pte_phys(physpage, pgprot)   __mk_pte((physpage) >> PAGE_SHIFT, pgprot)
```

https://elixir.bootlin.com/linux/2.4.22/source/include/asm-i386/pgtable.h#L312

The input parameters are:

- A frame physical address physpage, of type `unsigned long`
- A bit string pgprot for a PTE, of type `pgprot_t`

The macro builds a complete PTE entry, which includes the physical address of the target frame. The return type is `pte_t` and it can be then assigned to one PTE entry.

# Loading the page table

V2.4

```
344  /*
345   * paging_init() sets up the page tables - note that the first 8MB are
346   * already mapped by head.S.
347   *
348   * This routines also unmaps the page at virtual kernel address 0, so
349   * that we can trap those pesky NULL-reference errors in the kernel.
350   */
351  void __init paging_init(void)
352  {
353          pagetable_init();
354
355          load_cr3(swapper_pg_dir);

365
366          __flush_tlb_all();
367
368  #ifdef CONFIG_HIGHMEM
369          kmap_init();
370  #endif
371          zone_sizes_init();
372  }
```

https://elixir.bootlin.com/linux/2.4.22/source/arch/i386/mm/init.c#L351

# load_cr3()

V2.4

```
184   #define load_cr3(pgdir) \
185           asm volatile("movl %0,%%cr3": :"r" (__pa(pgdir)));
```

https://elixir.bootlin.com/linux/2.4.22/source/include/asm-i386/processor.h#L184

The load cr3 instruction is directly mapped to the assembly code which loads the address of the PGD into CR3.

V5.11

Latest versions of the kernel uses the **paravirtualization** scheme to map all of the basic functions that for example regards the mmu, like writing CR3, creating PGD, PTE and so on. So in modern kernel you will find load_cr3 mapped to

```
138   static inline void write_cr3(unsigned long x)
139   {
140           PVOP_VCALL1(mmu.write_cr3, x);
141   }
```

https://elixir.bootlin.com/linux/v5.12-rc2/source/arch/x86/include/asm/paravirt.h#L138

# Final virtual kernel memory map

This is the final virtual **memory** map of the kernel, it has nothing to do with the physical counterpart.

Page tables created by `pagetable_init()` and allocated with bootmem

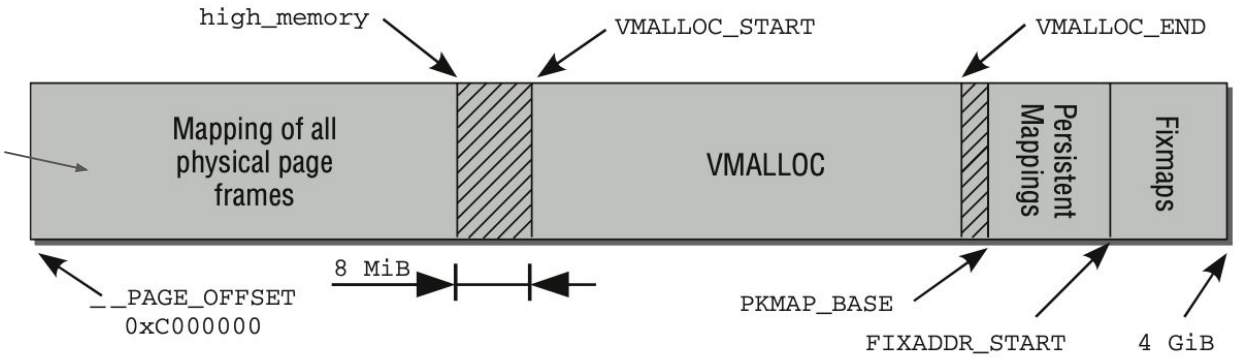



**Figure 3-15: Division of the kernel address space on IA-32 systems.**

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

As you can see the kernel may only address only 896MB because the last 128MB are reserved:

- **VMALLOC** - virtual contiguous memory areas that are not contiguous in physical memory, especially for user processes.
- **persistent mappings** - used for mapping highmem pages
- **fixmaps** - virtual addresses customly mapped to selectable physical frames

# 2.3.5

## 2. Step 4: Kernel Boot

3. start_kernel()

# TLB



DIAG

# Implicit and explicit operations

The degree of automation in the management process of TLB entries depends on the hardware architecture. Kernel hooks exist for explicit management of TLB operations (mapped at compile time to nops in case of fully-automated TLB management)

On x86, automation is only **partial**: automatic TLB flushes occur upon updates of the $CR_3$ register (e.g. page table changes) but Changes inside the current page table are not automatically reflected into the TLB.

# TLB Relevant events

## Scale classification

- **global**: dealing with virtual addresses accessible by every CPU/core in real-time-concurrency
- **local**: dealing with virtual addresses accessible in timesharing concurrency

## Typology classification

- Virtual to physical address remapping
- Virtual address access rule modification (read only vs write access)

The typical management is TLB implicit renewal via flush operations

# TLB Flush Costs

## Direct costs

- the latency of the firmware level protocol for TLB entries invalidation (selective vs non-selective)
- the latency for cross-CPU coordination in case of global TLB flushes

## Indirect costs

- TLB renewal latency by the MMU firmware upon misses in the translation process of virtual to physical addresses and this cost depends on the amount of entries to be refilled
- Tradeoff vs TLB API and software complexity inside the kernel (selective vs non-selective flush/renewal)

# Linux full TLB flush

V5.12

```
929  void flush_tlb_all(void)
930  {
931          count_vm_tlb_event(NR_TLB_REMOTE_FLUSH);
932          on_each_cpu(do_flush_tlb_all, NULL, 1);
933  }
```

https://elixir.bootlin.com/linux/v5.12-rc2/source/arch/x86/mm/tlb.c#L929

This flushes the **entire TLB** on **all** processors running in the system (most expensive TLB flush operation). After it completes, all modifications to the page tables are globally visible. This is required after the kernel page tables, which are global in nature, have been modified.

# Linux TLB Flush

V2.4

```
38  #define __flush_tlb()                                                   \
39          do {                                                            \
40                  unsigned int tmpreg;                                    \
41                                                                          \
42                  __asm__ __volatile__(                                   \
43                          "movl %%cr3, %0;  # flush TLB \n"               \
44                          "movl %0, %%cr3;              \n"               \
45                          : "=r" (tmpreg)                                 \
46                          :: "memory");                                   \
47          } while (0)
48
49  /*
50   * Global pages have to be flushed a bit differently. Not a real
51   * performance problem because this does not happen often.
52   */
53  #define __flush_tlb_global()                                            \
54          do {                                                            \
55                  unsigned int tmpreg;                                    \
56                                                                          \
57                  __asm__ __volatile__(                                   \
58                          "movl %1, %%cr4;  # turn off PGE     \n"        \
59                          "movl %%cr3, %0;  # flush TLB        \n"        \
60                          "movl %0, %%cr3;                     \n"        \
61                          "movl %2, %%cr4;  # turn PGE back on \n"        \
62                          : "=&r" (tmpreg)                                \
63                          : "r" (mmu_cr4_features & ~X86_CR4_PGE),        \
64                            "r" (mmu_cr4_features)                        \
65                          : "memory");                                    \
66          } while (0)
67
```

# Linux partial TLB flush

```
void flush_tlb_mm(struct mm_struct *mm)
```

This flushes all TLB entries related to a **portion** of the userspace memory context. On some architectures (e.g. MIPS), this is required for all cores (usually it is confined to the local processor).

This is called only after an operation affecting the entire address space:

- when cloning a process with a fork()
- when, in general, there is an interaction with the Copy-On-Write protection

```
void flush_tlb_page(struct vm_area_struct *vma, unsigned long a);
```

This API flushes a single page from the TLB. The two most common uses of it are to flush the TLB after a page has been faulted in or has been paged out.

# Linux partial TLB flush

```
void flush_tlb_range(struct mm_struct *mm, unsigned long start,unsigned long end);
```

This flushes all entries within the requested user space range for the mm context. This is used after a region has been moved (mremap()) or when changing permissions (mprotect()). This API is provided for architectures that can remove ranges of TLB entries quicker than iterating with `flush_tlb_page()`.

```
void flush_tlb_pgtables(struct mm_struct *mm, unsigned long start,
                        unsigned long end);
```

Used when the page tables are being torn down and free'd. Some platforms cache the lowest level of the page table, which needs to be flushed when the pages are being deleted (e.g. Sparc64). This is called when a region is being unmapped and the page directory entries are being reclaimed.

# Linux partial TLB flush

```
void update_mmu_cache(struct vm_area_struct *vma, unsigned long addr, pte_t *ptep);
```

Only called after a page fault completes. It tells that a new translation now exists at `pte` for the virtual address `addr`. Each architecture decides how this information should be used.

For example, Sparc64 uses the information to decide if the local CPU needs to flush its data cache. In some cases it is also used for preloading TLB entries

2.3.6

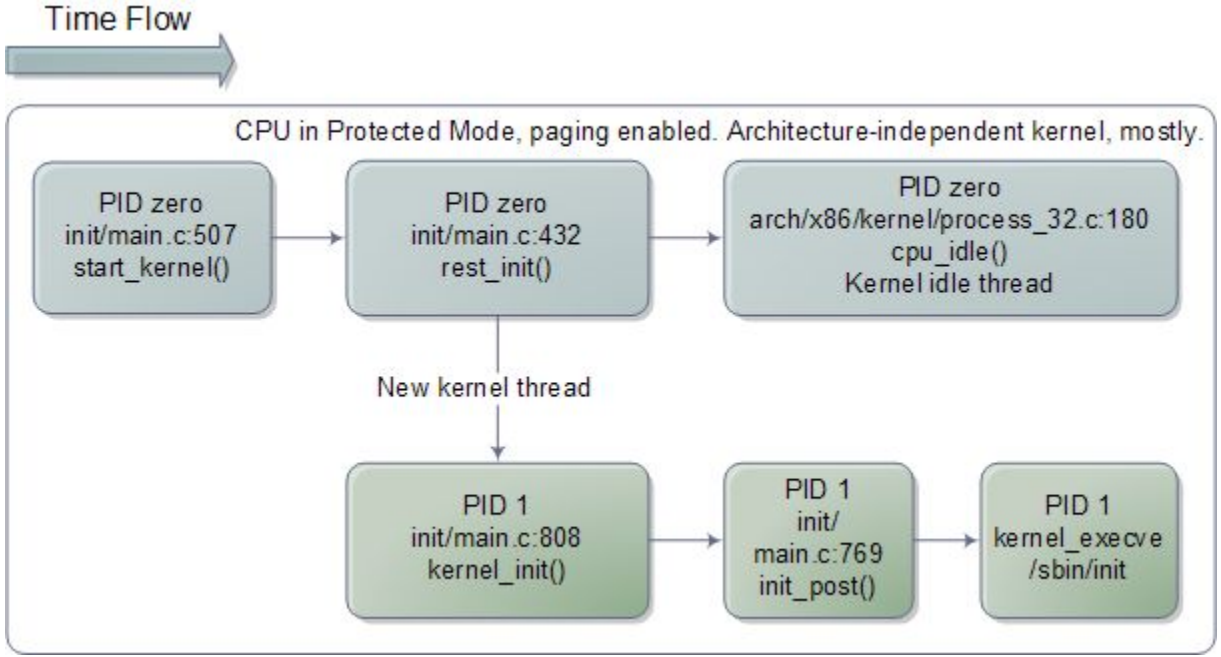
2. Step 4: Kernel Boot
3. start_kernel()

# Final Operations and Recap



DIAG

# Kernel Boot Flow



# Main operations

The main operations carried out by `start_kernel()` (`init/main.c`) are:

1. `setup_arch()` that initializes the architecture
2. `build_all_zonelists()` - builds the memory zones
3. `page_alloc_init()` / `mem_init()` - the steady state allocator (Buddy System) is initialized and the boot one removed
4. `sched_init()` - initializes the scheduler
5. `trap_init()` - the final IDT is built
6. `time_init()` - the system time is initialized
7. `kmem_cache_init()` - the slab allocator is initialized
8. `arch_call_rest_init()` / `rest_init()` - prepares the environment
   a. `kernel_thread(kernel_init)` - starts the kernel thread for process 1 is created
      i. `kernel_init_freeable()` -> `prepare_namespace()` -> `initrd_load()` - mounts the initramfs, a temporary filesystem used to start the init process
      ii. `run_init_process()` -> `kernel_execve()` - Execute `/bin/init`
   b. `cpu_startup_entry()` -> `do_idle()` - starts the idle process

# Final GDT

| Linux's GDT | Segment Selectors |
|---|---|
| null | 0x0 |
| reserved | |
| reserved | |
| reserved | |
| not used | |
| not used | |
| TLS #1 | 0x33 |
| TLS #2 | 0x3b |
| TLS #3 | 0x43 |
| reserved | |
| reserved | |
| reserved | |
| kernel code | 0x60 (__KERNEL_CS) |
| kernel data | 0x68 (__KERNEL_DS) |
| user code | 0x73 (__USER_CS) |
| user data | 0x7b (__USER_DS) |

| Linux's GDT | Segment Selectors |
|---|---|
| TSS | 0x80 |
| LDT | 0x88 |
| PNPBIOS 32-bit code | 0x90 |
| PNPBIOS 16-bit code | 0x98 |
| PNPBIOS 16-bit data | 0xa0 |
| PNPBIOS 16-bit data | 0xa8 |
| PNPBIOS 16-bit data | 0xb0 |
| APMBIOS 32-bit code | 0xb8 |
| APMBIOS 16-bit code | 0xc0 |
| APMBIOS data | 0xc8 |
| not used | |
| not used | |
| not used | |
| not used | |
| not used | |
| double fault TSS | 0xf8 |

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

# Idle Process

```
391  void cpu_startup_entry(enum cpuhp_state state)
392  {
393          arch_cpu_idle_prepare();
394          cpuhp_online_idle(state);
395          while (1)
396                  do_idle();
397  }
```

https://elixir.bootlin.com/linux/v5.11.2/source/kernel/sched/idle.c#L391

```
256  /*
257   * Generic idle loop implementation
258   *
259   * Called with polling cleared.
260   */
261  static void do_idle(void)
262  {
263          int cpu = smp_processor_id();

276          while (!need_resched()) {
277                  rmb();
278
279                  local_irq_disable();
280

295                  if (cpu_idle_force_poll || tick_check_broadcast_expired
296                          tick_nohz_idle_restart_tick();
297                          cpu_idle_poll();
298                  } else {
299                          cpuidle_idle_call();
300                  }
301                  arch_cpu_idle_exit();
302          }

327          schedule_idle();

331  }
```

```
675  /*
676   * Called from the generic idle code.
677   */
678  void arch_cpu_idle(void)
679  {
680          x86_idle();
681  }
```

Paravirt. (newer kernels)

```
148  static inline void arch_safe_halt(void)
149  {
150          PVOP_VCALL0(irq.safe_halt);
151  }
```

```
static inline void native_halt(void) {
        asm volatile("hlt": : :"memory");
}
```

# End of the Kernel Boot

The idle loop is the **ending** of the kernel booting process.

Since the very first long jump `ljmp $0xf000,$0xe05b` at the reset vector at `F000:FFF0` which activated the BIOS, we have worked hard to setup a system which is spinning forever.

This is the end of the "romantic" Kernel boot procedure: we infinitely loop into a hlt instruction or ...

# Advanced Operating Systems and Virtualization

## [2] Step 4: Kernel Boot

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/



gpm.name · proiettimattia@diag.uniroma1.it

DIAG