Linux Kernel Boot

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2019/2020

Sapienza
Università di Roma

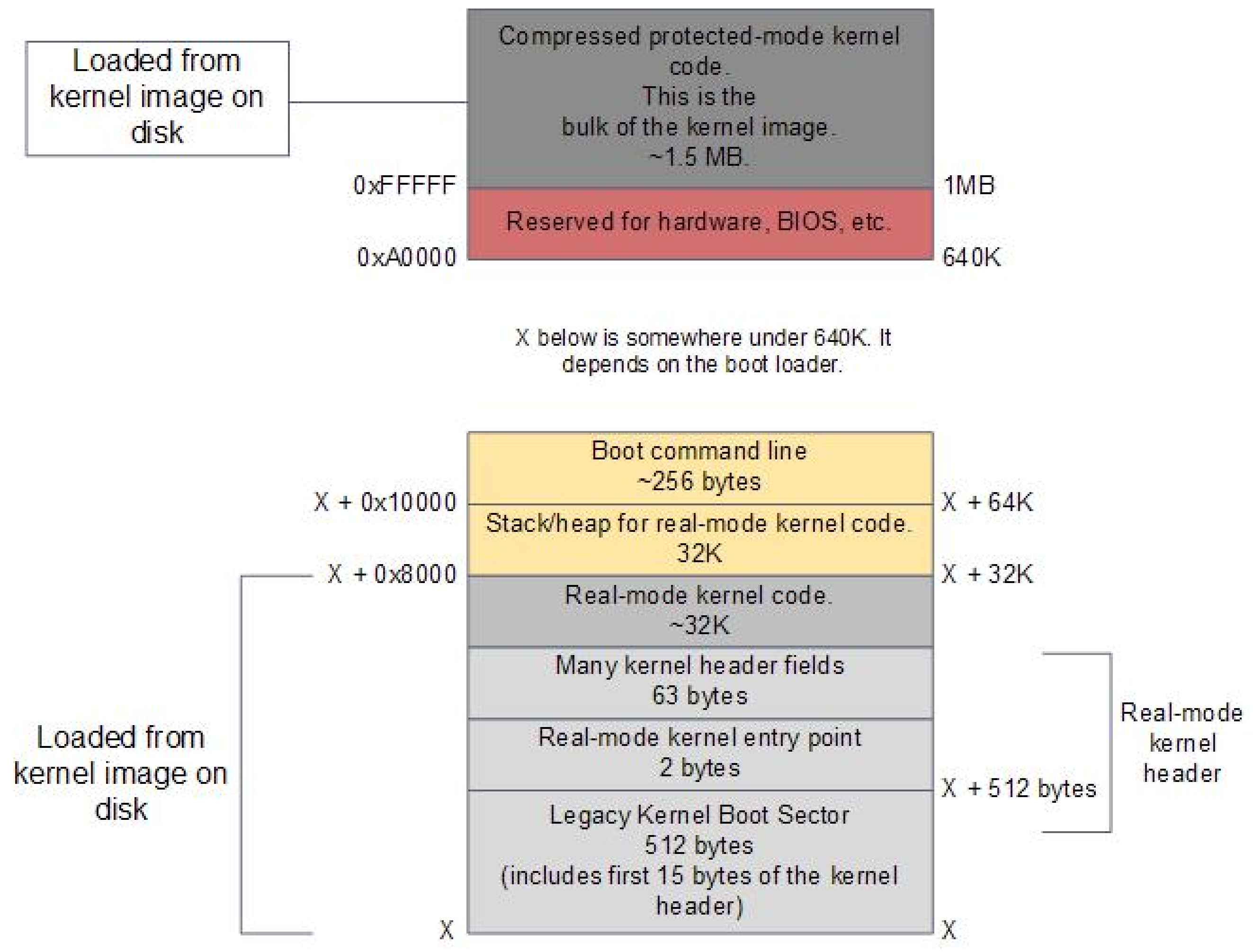
Boot Sequence

BIOS/UEFI	The actual Hardware Startup
Bootloader Stage 1	Executes the Stage 2 bootloader (skipped in case of UEFI)
Bootloader Stage 2	Loads and starts the Kernel
Kernel Startup	**The Kernel takes control of and initializes the machine (machine-dependent operations)**
Init	First process: basic environment initialization (e.g., SystemV Init, systemd)
Runlevels/Targets	Initializes the user environment (e.g., single-user mode, multiuser, graphical, ...)

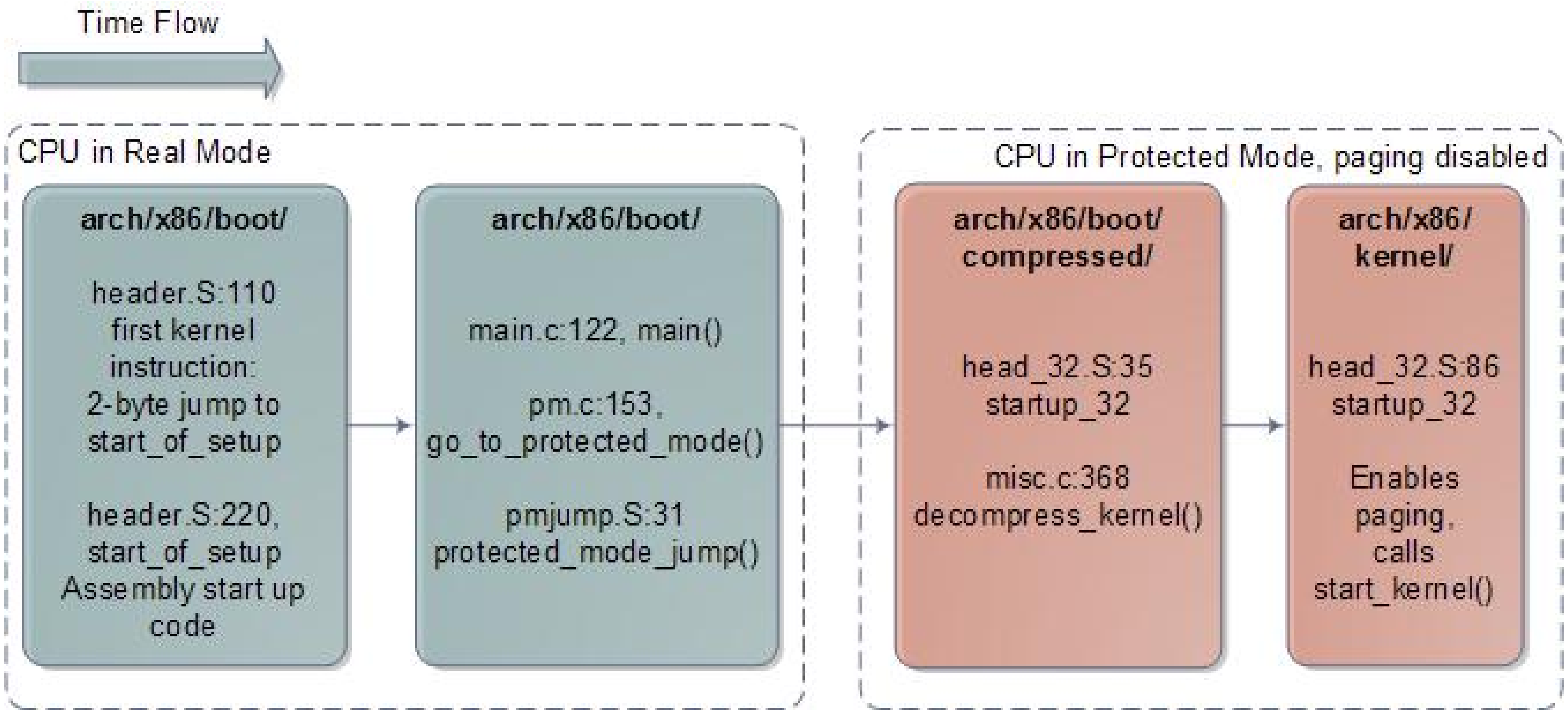
Initial Life of the Linux Kernel

- The Second stage bootloader (or the UEFI bootloader) loads the initial image of the kernel in memory
- This kernel image is very different from the steady-state one
- The entry point of the kernel must be identified by the bootloader

RAM after the bootloader is done



Initial Life of the Linux Kernel



References to code are related to Linux 2.6.24

In newer versions, the flow is the same, but line numbers change

Initial Life of the Linux Kernel

- The early kernel start-up for the Intel architecture is in file `arch/x86/boot/header.S`
- The very first executed instruction is at `_start`:

```
_start:
        .byte   0xeb            # short (2-byte) jump
        .byte   start_of_setup-1f
1:
... (around 300 lines of data and support routines)
start_of_setup:
```

start_of_setup()

- This short routine makes some initial setup:
  - It sets up a stack
  - It zeroes the bss section (just in case...)
  - It then jumps to `main()` in `arch/x86/boot/main.c`
- Here the kernel is still running in real mode
- This function implements part of the the *Kernel Boot Protocol*
- This is the moment when boot options are loaded in memory

main()

- After some housekeeping and sanity checks, main() calls go_to_protected_mode() in arch/x86/boot/pm.c
- The goal of this function is to prepare the machine to enter protected mode and then do the switch
- This follows exactly the steps which we discussed:
  - Enabling A20 line
  - Setting up Interrupt Descriptor Table
  - Setup memory

Interrupt Descriptor Table

- In real mode, the *Interrupt Vector Table* is always at address zero
- We now have to load the IDT into the IDTR register. The following code ignores all interrupts:

```
static void setup_idt(void)
{
        static const struct gdt_ptr null_idt = {0, 0};
        asm volatile("lidtl %0" : : "m" (null_idt));
}
```

setup_gdt()

```
static void setup_gdt(void)
{
        static const u64 boot_gdt[] __attribute__((aligned(16))) = {
                [GDT_ENTRY_BOOT_CS] = GDT_ENTRY(0xc09b, 0, 0xfffff),
                [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0, 0xfffff),
                [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096, 103),
        };

        static struct gdt_ptr gdt;
        gdt.len = sizeof(boot_gdt)-1;
        gdt.ptr = (u32)&boot_gdt + (ds() << 4);

        asm volatile("lgdtl %0" : : "m" (gdt));
}
```

`GDT_ENTRY` **is defined as a macro in** `arch/x86/include/asm/segment.h`

Moving to protected 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 is an assembly routine which:
  - Sets the PE bit in CR0 (paging still disabled)
  - Issues a `ljmp` to its very next instruction to load in CS the boot CS selector
  - Sets up data segments for flat 32-bit mode
  - It sets a (temporary) stack

Decompressing the Kernel

- `protected_mode_jump()` jumps into `startup_32()` in `arch/x86/boot/compressed/head_32.S`
- This routine does some basic initialization:
  - 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
  - Calls `decompress_kernel()` (or `extract_kernel()`) in `arch/x86/boot/compressed/misc.c`

Kernel Address Space Layout Randomization (KASLR)

- If you know the binary image of the kernel, an attacker patch the memory image of the kernel by writing directly at the correct address in memory
- At boot time, the kernel "randomly" decides where to decompress itself in memory, relying on the most accurate source of entropy available
- The number of possibilities is anyhow reduced:
  - The kernel is mapped using 2MB pages
  - The number of "vaild slots" is thus limited

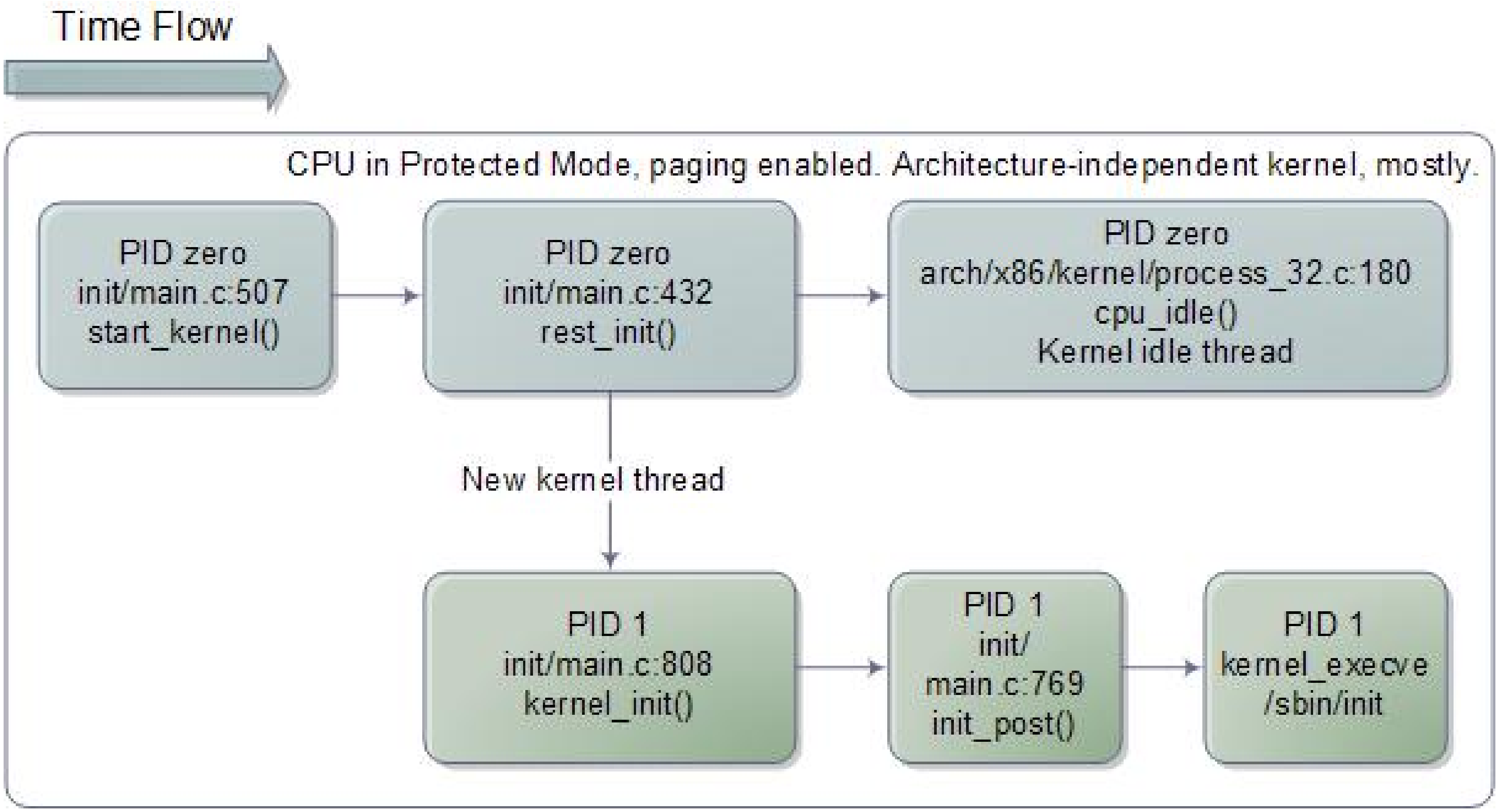
(Actual) Kernel entry point

- The first startup routine of the decompressed kernel is `startup_32()` at `arch/x86/kernel/head_32.S`
- Here we start to prepare the final image of the kernel which will be resident in memory until we shut down the machine
- Remember that paging is still disabled!

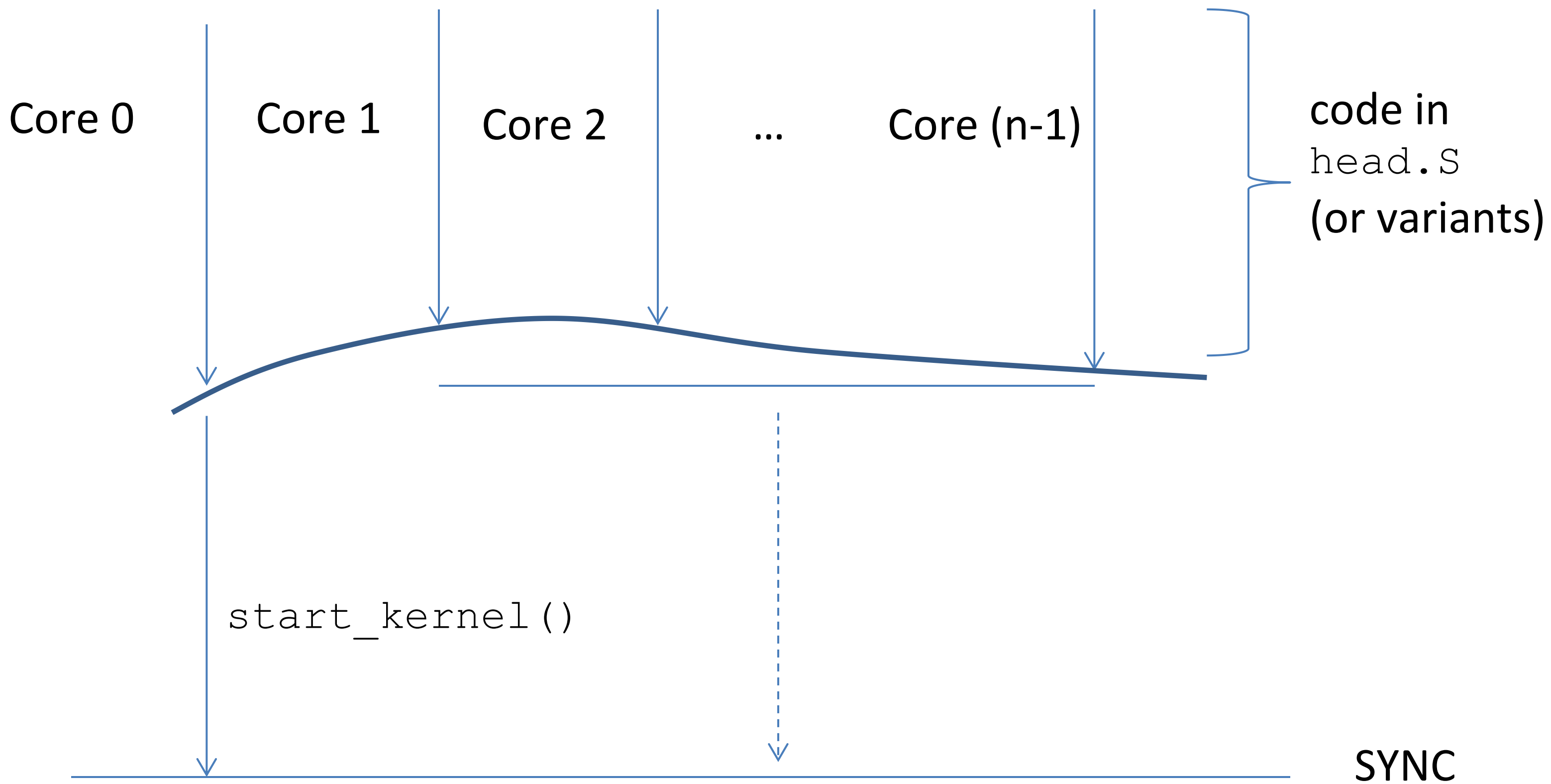
startup_32() (second version)

- Clear the BSS segment again
- Setup a new GDT
- Build the page table
- Enable paging
- Create the final IDT
- Jump into the architecture-independent kernel entry point (`start_kernel()` at `init/main.c`)

Kernel Initialization



Kernel Initialization



Kernel Initialization

- `start_kernel()` executes on a single core (master)
- All the other cores (slaves) keep waiting that the master has finished
- The kernel internal function `smp_processor_id()` can be used to retrieve the ID of the current core
- It is based on ASM instructions implementing a hardware specific ID detection protocol
- On newer versions, it reads the CPU ID from APIC
- This function can be used both at kernel startup and at steady state

Inline Assembly

```
__asm__ __volatile__ (
        Assembly Template
        : Output Operands
        : Input Operands
        : Clobbers
);
```

A string keeping one or more assembly instructions

A comma-separated list of inputs
"=r" (old), "+rm" (*Base)

A comma-separated list of outputs
"r" (Offset)

A comma-separated list of registers or other elements changed by the execution of the instruction(s)

Inline Assembly

- "m": a memory operand
- "o": a memory operand which is "offsettable" (to deal with instructions' size)
- "r": a general-purpose register
- "g": Register, memory or immediate, except for non-general purpose registers
- "i": an immediate operand
- "0", "1", … ‘9’: a previously referenced register
- "q": any "byte-addressable" register
- “Q” any “high” 8-bit addressable sub-register
- "+": the register is both read and written
- "=": the register is written
- "a", "b", "c", "d", "S", "D": registers A, B, C, D, SI, and DI
- "A": registers A and D (for instructions using AX:DX as output)

CPUID Identification

- When available, the `cpuid` assembly instruction gives information about the available hardware

```
void cpuid(int code, uint32_t *a, uint32_t *d) {
      asm volatile("cpuid"
      :"=a"(*a),"=d"(*d)
      :"a"(code)
      :"ecx","ebx");
}
```

wrmsr/rdmsr

```
static inline void wrmsr(uint32_t msr_id, uint64_t msr_value)
{
    asm volatile ( "wrmsr" : : "c" (msr_id), "A" (msr_value) );
}

static inline uint64_t rdmsr(uint32_t msr_id)
{
    uint64_t msr_value;
    asm volatile ( "rdmsr" : "=A" (msr_value) : "c" (msr_id) );
    return msr_value;
}
```

Kernel Initialization Signature

- `start_kernel()` is declared as:

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

- `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`: free this memory after initialization (maps to a specific section)

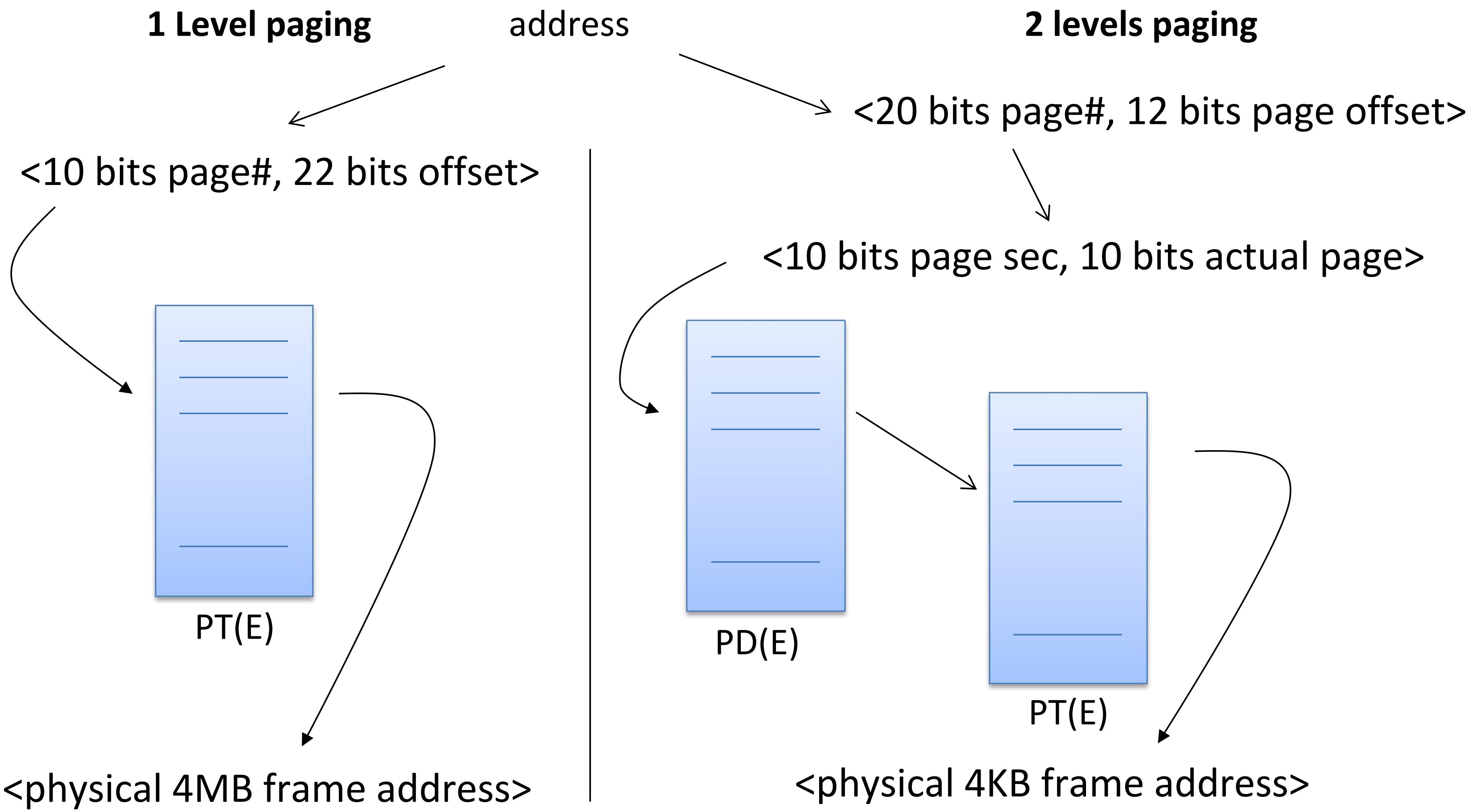
Some facts about memory

- During initialization, the steady-state kernel must take control of the available physical memory (see `setup_arch()` at `kernel/setup.c`)
- This is due to the fact that it will have to manage it with respect to virtual address spaces of all processes
  - Memory allocation and deallocation
  - Swapping
- When starting, the kernel must have an early organization setup out of the box

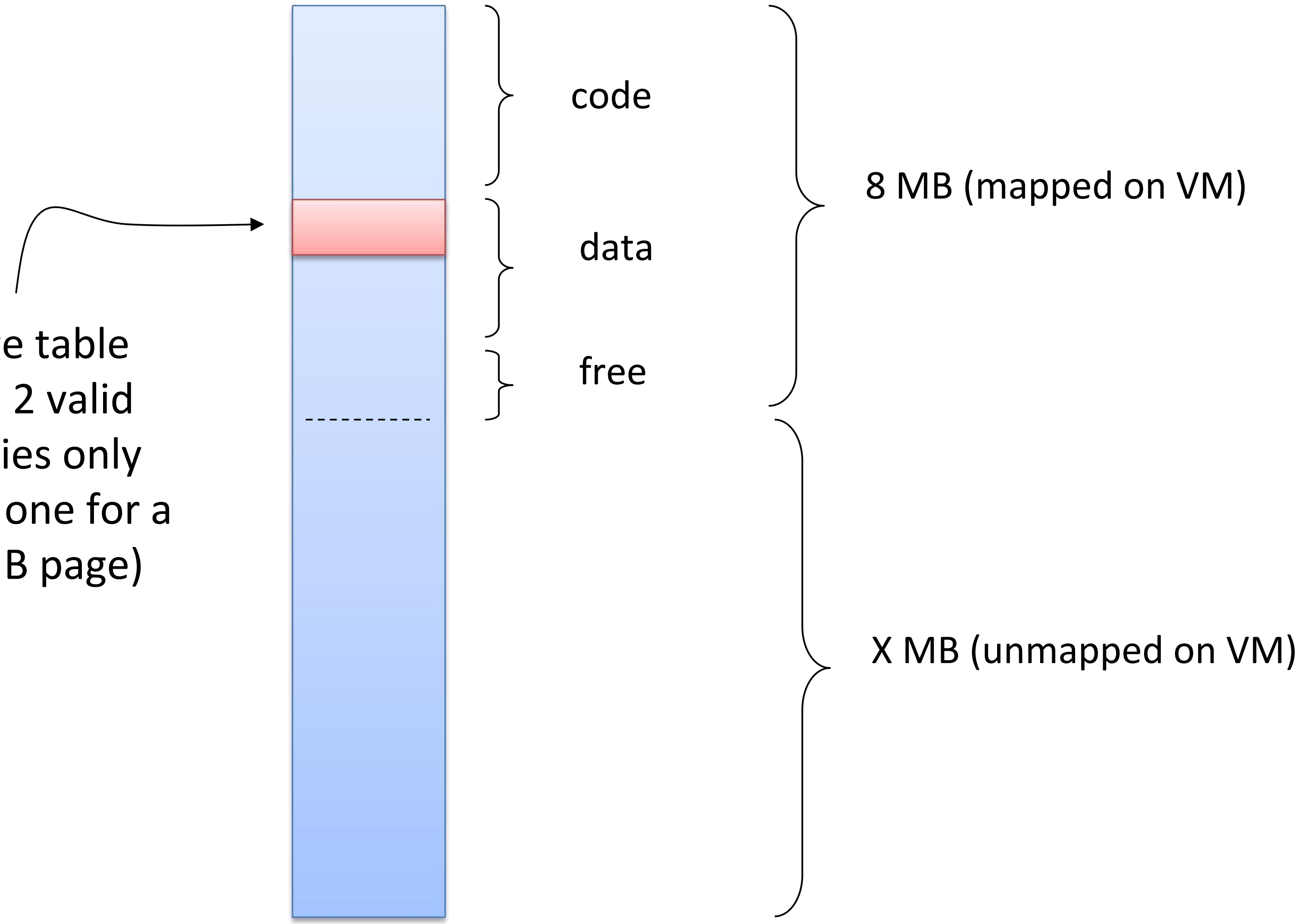
Enabling Paging

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

Early Page Table Organization (i386)



Early Page Table Organization (i386)



Page table
with 2 valid
entries only
(each one for a
4 MB page)

What do we have to do now

1. We need to reach the correct granularity for paging (4KB)
2. We need to span logical to physical address across the whole 1GB of manageable physical memory
3. We need to re-organize the page table in two separate levels
4. So we need to determine 'free buffers' within the already reachable memory segment to initially expand the page table
5. We cannot use memory management facilities other than paging (Kernel-level memory manager is not ready yet!)
6. We need to find a way to describe the physical memory
7. We're not dealing with userspace memory yet!

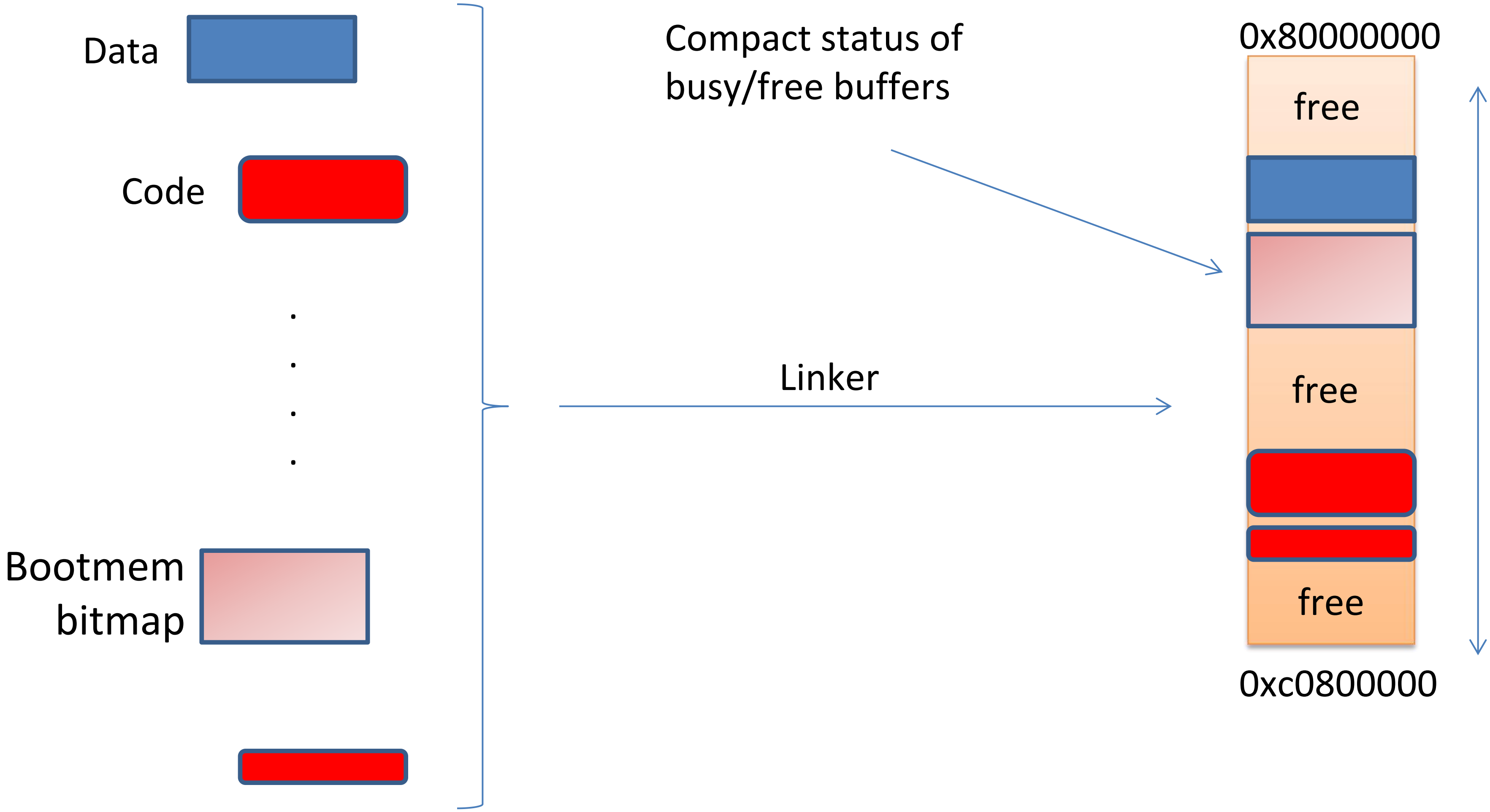
Kernel-Level MM Data Structures

- Kernel Page table
  - It keeps the memory mapping for kernel-level code and data (thread stack included)
- Core map
  - The map that keeps status information for any frame (page) of physical memory, and for any NUMA node
  - Free list of physical memory frames, for any NUMA node

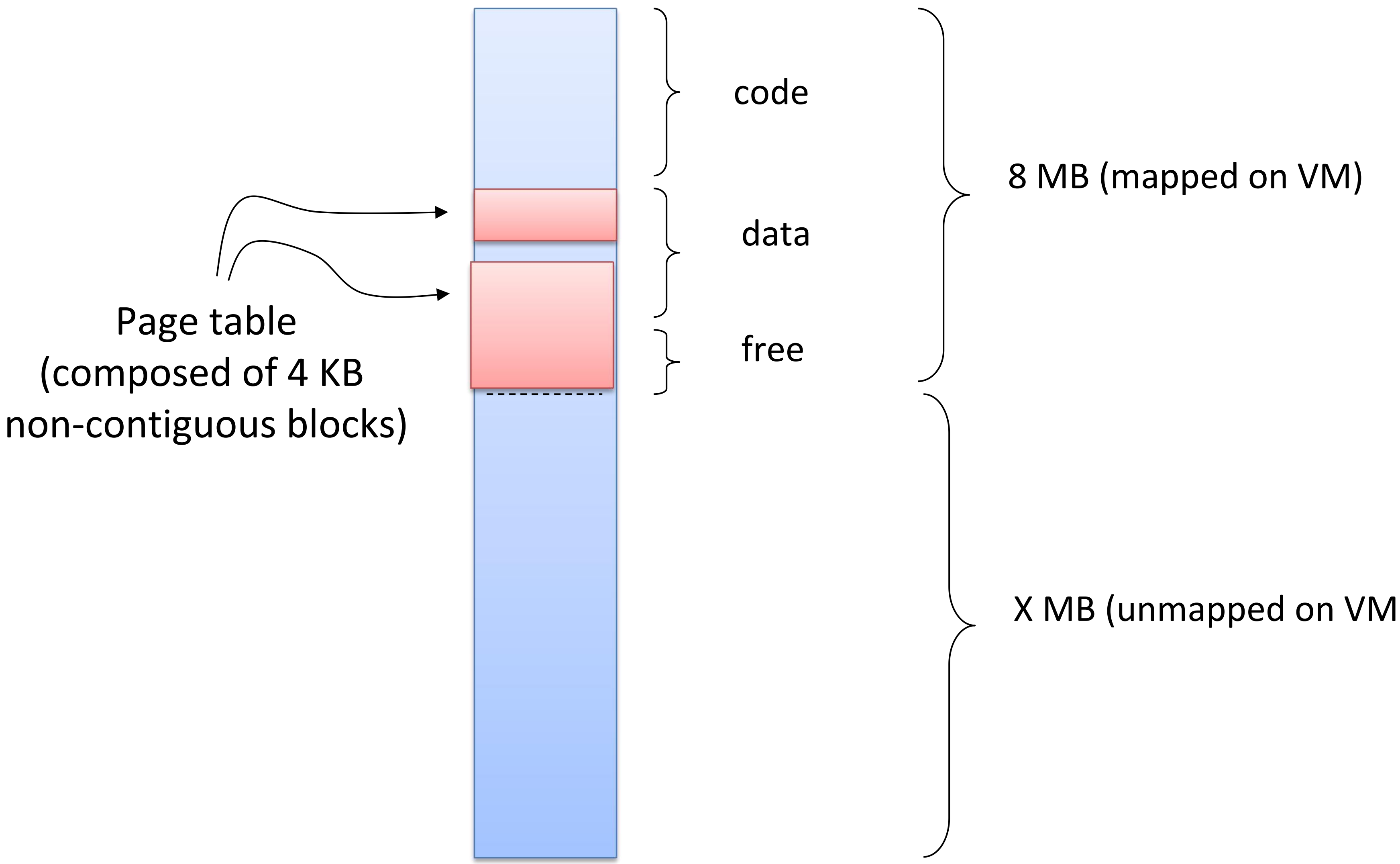
Bootmem

1. Memory map of the initial kernel image is known at compile time
2. A link time memory manager is embedded into the kernel image, which is called *bootmem allocator* (see `linux/bootmem.h`)
3. It relies on bitmaps telling if any 4KB page in the currently reachable memory image is busy or free
4. It also offers API (at boot time only) to get free buffers
5. These buffers are sets of contiguous page-aligned areas

Bootmem organization



Location of PT in Physical Memory

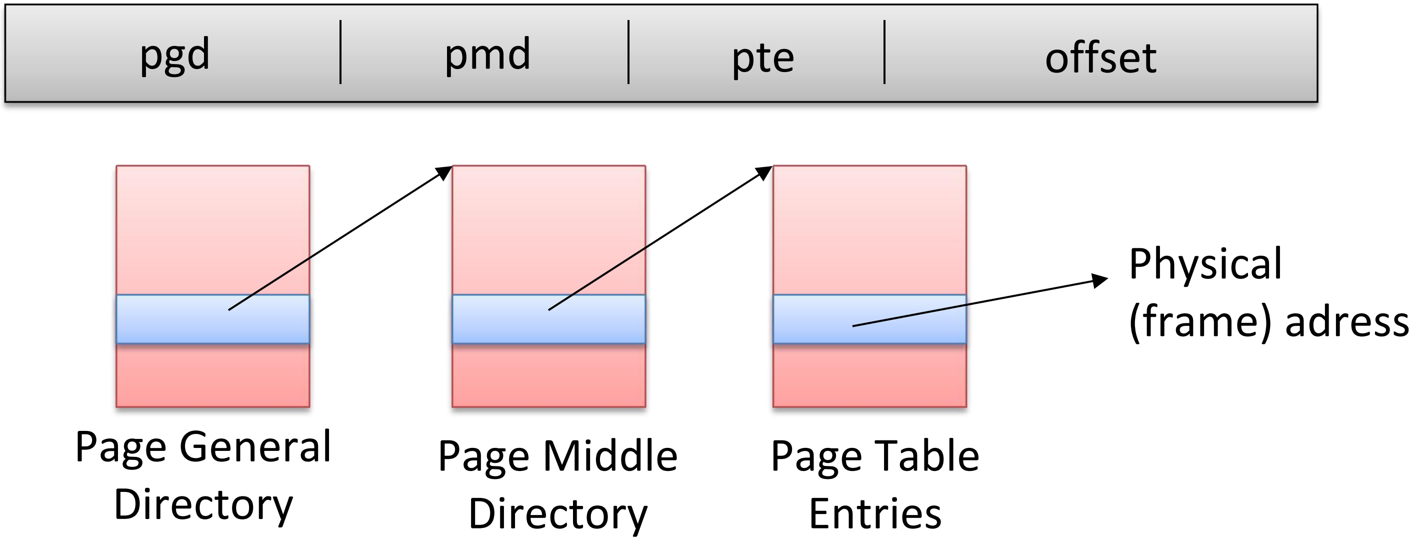


Memblock

- The Logical Memory Block (LMB) allocator has superseded Bootmem on almost all architectures
- The idea behind it is that available memory is larger and addressing is more scattered
- Memory is represented as two arrays of regions
  - Physically-contiguous memory
  - Allocated regions
- `memblock_add[_node]()`: it registers a physical memory range
- `memblock_reserve()`: mark a range of memory as busy
- `memblock_find_in_range()`: find an (aligned) free area in given range

How Linux handles paging

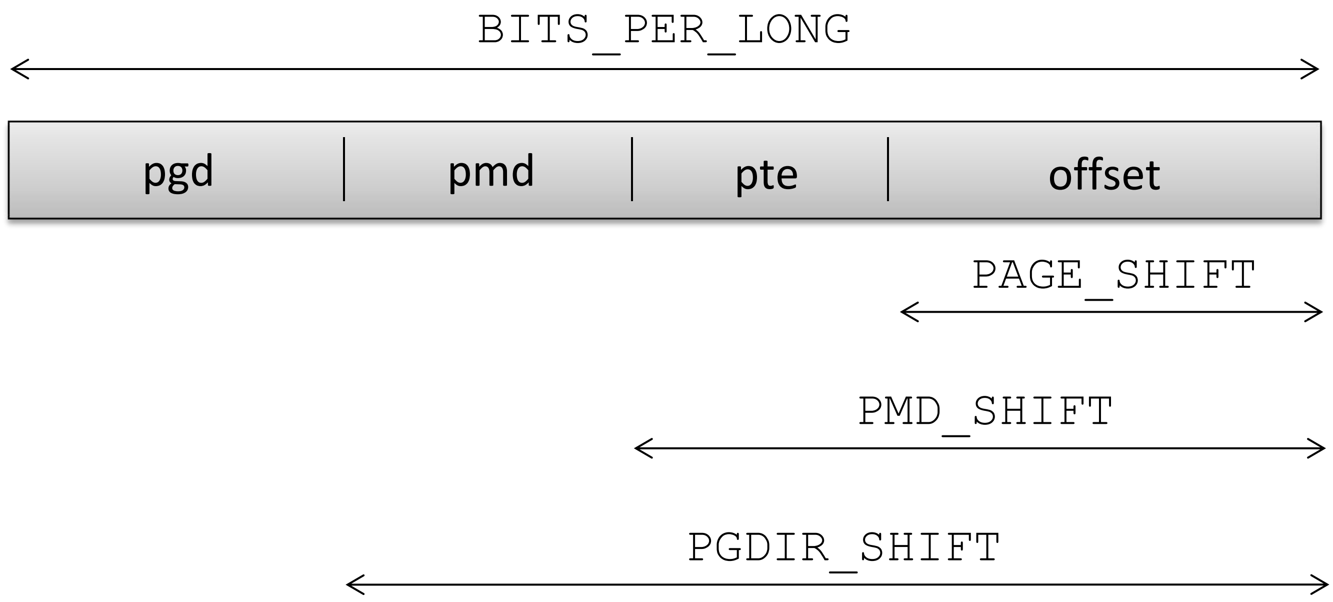
- Linux on x86 has 3 indirection levels:



- Linux has also the possibility to manage 4 levels:
  - Page Global Directory, Page Upper Directory, Page Middle Directory, Page Table Entry

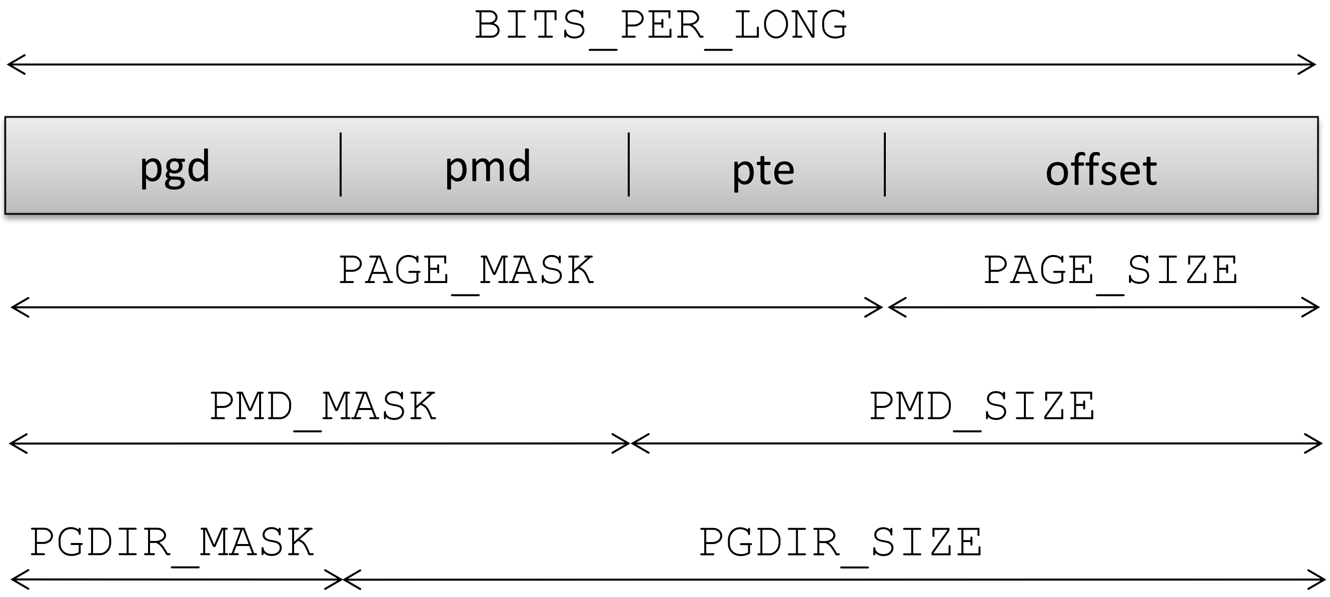
Splitting the address

- SHIFT macros specify the length in bit mapped to each PT level:
  - arch/x86/include/asm/pgtable-3level_types.h
  - arch/x86/include/asm/pgtable-2level_types.h
  - arch/x86/include/asm/page_types.h
  - arch/x86/include/asm/pgtable_64_types.h



Splitting the address

- MASK macros are used to retrieve higher bits
- SIZE macros reveal how many bytes are addressed by each entry



Configuring the PT

- There are the `PTRS_PER_x` macros which determine the number of entries in each level of the page table

```
#define PTRS_PER_PGD    1024
#define PTRS_PER_PMD    1      <-- without PAE
#define PTRS_PER_PTE    1024
```

Page Table Data Structures

- `swapper_pg_dir` in `arch/i386/kernel/head.S` keeps the virtual memory address of the PGD (PDE) portion of the kernel page table
- It is initialized at compile time, depending on the memory layout defined for the kernel bootable image
- Any entry within the PGD is accessed via displacement
- C types for the definition of the content of the page table entries are defined:

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

Fighting againts weak typing

- C is *weak typed*
- This code generates no errors nor warnings:

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

Bit fields

- In `arch/x86/include/asm/pgtable_types.h` we find the definitions of the fields proper of page table entries

```
#define _PAGE_BIT_PRESENT  0 /* is present */
#define _PAGE_BIT_RW       1 /* writeable */
#define _PAGE_BIT_USER     2 /* userspace addressable */
#define _PAGE_BIT_PWT      3 /* page write through */
#define _PAGE_BIT_PCD      4 /* page cache disabled */
#define _PAGE_BIT_ACCESSED 5 /* accessed (raised by
CPU) */
#define _PAGE_BIT_DIRTY          6/* was written (raised
by CPU)*/
```

Bit fields and masks

```
pte_t x;

x = …;

if ((x.pte_low) & _PAGE_PRESENT){
     /* the page is loaded in a frame */
} else {
      /* the page is not loaded in any
            frame */
} ;
```

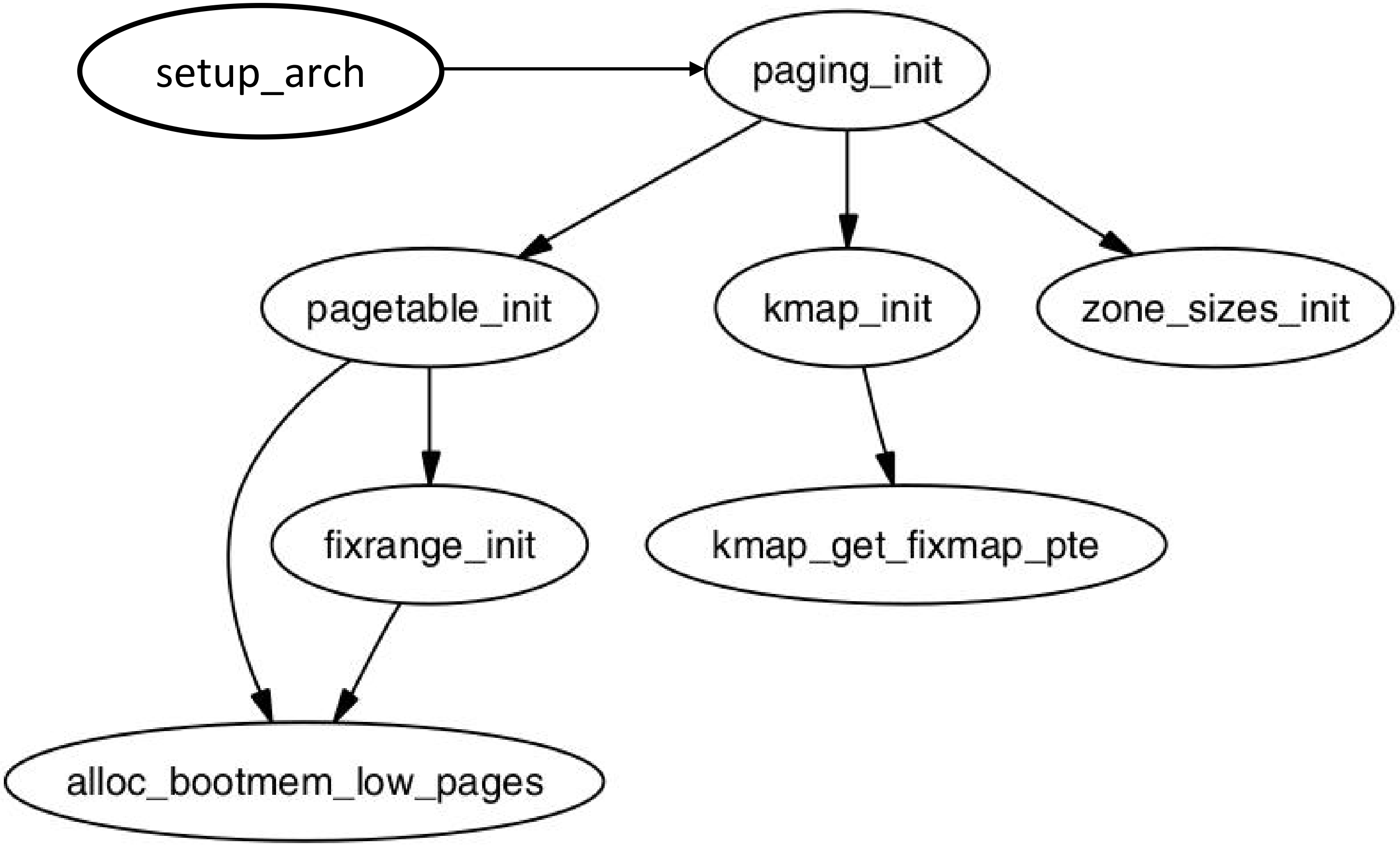
Different PD Entries

- Again 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)
```

Initialization Steps



Kernel Page Table Initialization

- As said, the kernel PDE is accessible at the virtual address kept by `swapper_pg_dir`
- PTEs are reserved within the 8MB of RAM accessible via the initial paging scheme
- Allocation done via `alloc_bootmem_low_pages()` defined in `include/linux/bootmem.h` (returns a virtual address)
- It returns the pointer to a page-aligned buffer with a size multiple of 4KBs

pagetable_init() (2.4.22)

```
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)));
        .........
    }
}
```

`pagetable_init()` (2.4.22)

- The final PDE buffer is the same as the initial page table mapping 4 MB pages
- 4KB paging is activated when filling the entry of the PDE table (Page Size bit is updated accordingly)
- Therefore, the PDE entry is set only after having populated the corresponding PTE table
- Otherwise memory mapping would be lost upon any TLB miss

__set_pmd() and __pa()

```
#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

```
#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 i386
  - The opposite can be done using the `__va(x)` macro

mk_pte_phys()

```
mk_pte_phys(physpage, pgprot)
```

- 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`
- The returned value can be then assigned to one PTE entry

Loading the new page table

- When `pagetable_init()` returns, the new page table is built
- The CPU is still relying on the boot pagetable
- Two lines in `paging_init()` make the new table visible to the architecture:

```
load_cr3(swapper_pg_dir);
__flush_tlb_all();
```

Invalidates Address-Space ID (ASID) on x86

load_cr3()

- in arch/x86/include/asm/processor.h:

```
static inline void load_cr3(pgd_t *pgdir)
{
    native_write_cr3(__pa(pgdir));
}
```

- in arch/x86/include/asm/special_insns.h:

```
static inline void native_write_cr3(unsigned long val) {
        asm volatile(
                "mov %0,%%cr3"
                :: "r" (val), "m" (__force_order)
        );
}
```

Dummy global variable to force serialization (better than memory clobber)

TLB implicit vs. 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 CR3 register (e.g. page table changes)
- Changes inside the current page table are not automatically reflected into the TLB

Types of 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 time-sharing concurrency
- **Typology** classification
  - Virtual to physical address remapping
  - Virtual address access rule modification (read only vs write access)
- Typical management: 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)
  - **plus**, 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
  - 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

```
void flush_tlb_all(void)
```

- 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 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)
- Called only after an operation affecting the entire address space
  - For example, when cloning a process with a `fork()`
  - Interaction with COW protection

Linux partial TLB flush

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

- 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
  - Interactions with page table access firmware

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()`

Linux partial TLB flush

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

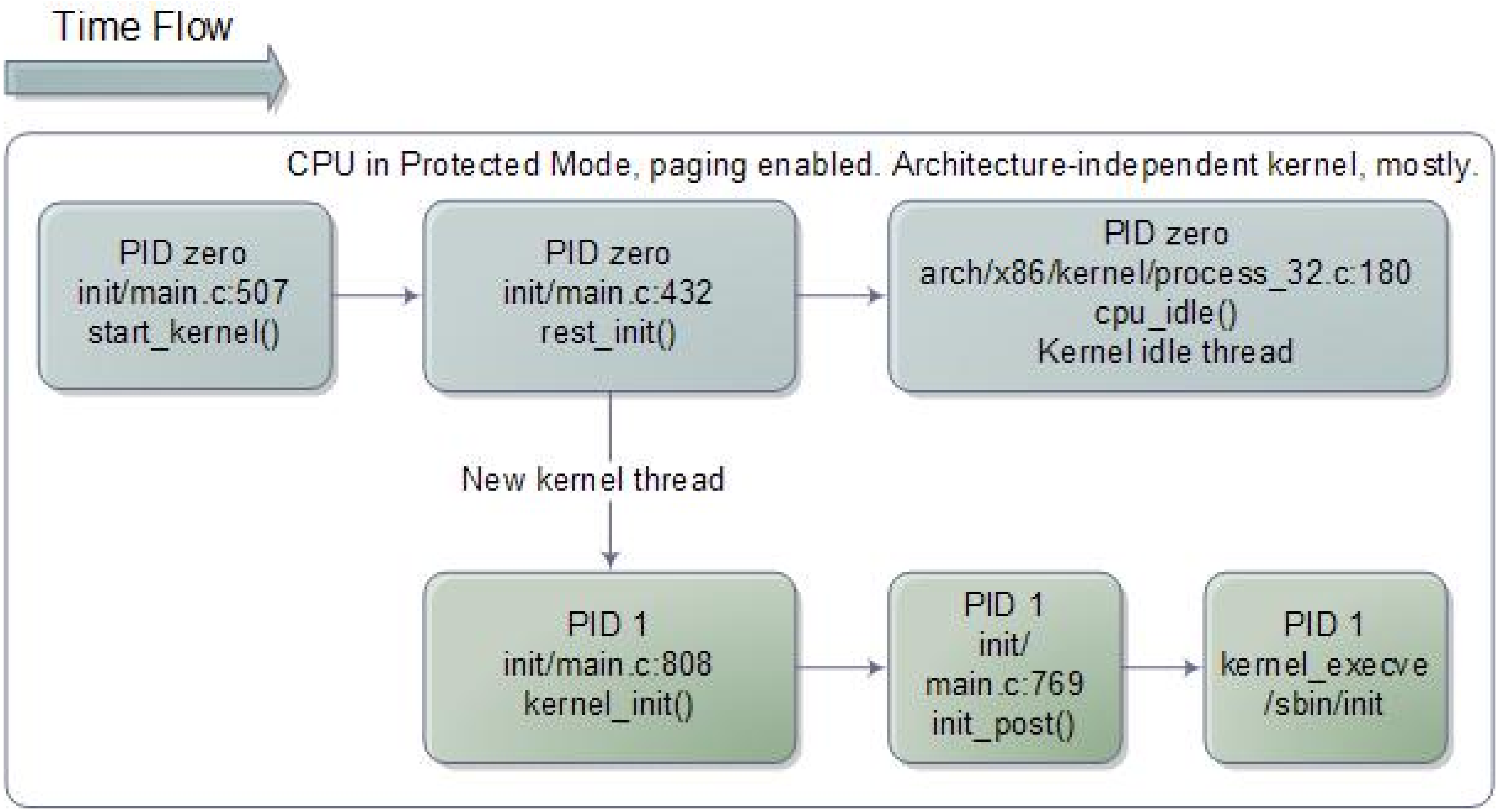
- Used when the page tables are being torn down and free'd
- Some platforms cache the lowest level of the page table, which needs to be flushed when the pages are being deleted (e.g. Sparc64)
- This is called when a region is being unmapped and the page directory entries are being reclaimed

Linux partial TLB flush

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

- 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*

Kernel Initialization



Setting up the Final GDT and IDT

- We have seen that during initialization, the kernel installs a dummy IDT:

```
static void setup_idt(void) {
        static const struct gdt_ptr null_idt = {0, 0};
        asm volatile("lidtl %0" : : "m" (null_idt));
}
```

- After having initialized memory, it's time to setup the final GDT and IDT
- In `start_kernel()`, after `setup_arch()` we find a call to `trap_init()` (defined in `arch/x86/kernel/traps.c`)

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	← Different for all cores
LDT	0x88	← Shared across all cores
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	

Per-core, instantiated at `arch/x86/kernel/cpu/common.c`

cpu_idle()

```
static void cpu_idle_loop(void) {
    while (1) {
        while(!need_resched()) {
            cpuidle_idle_call();
        }


         schedule_preempt_disabled();
    }
}

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

The End of the Booting Process

- The idle loop is the end of the 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...