Linux Kernel Boot

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2017/2018

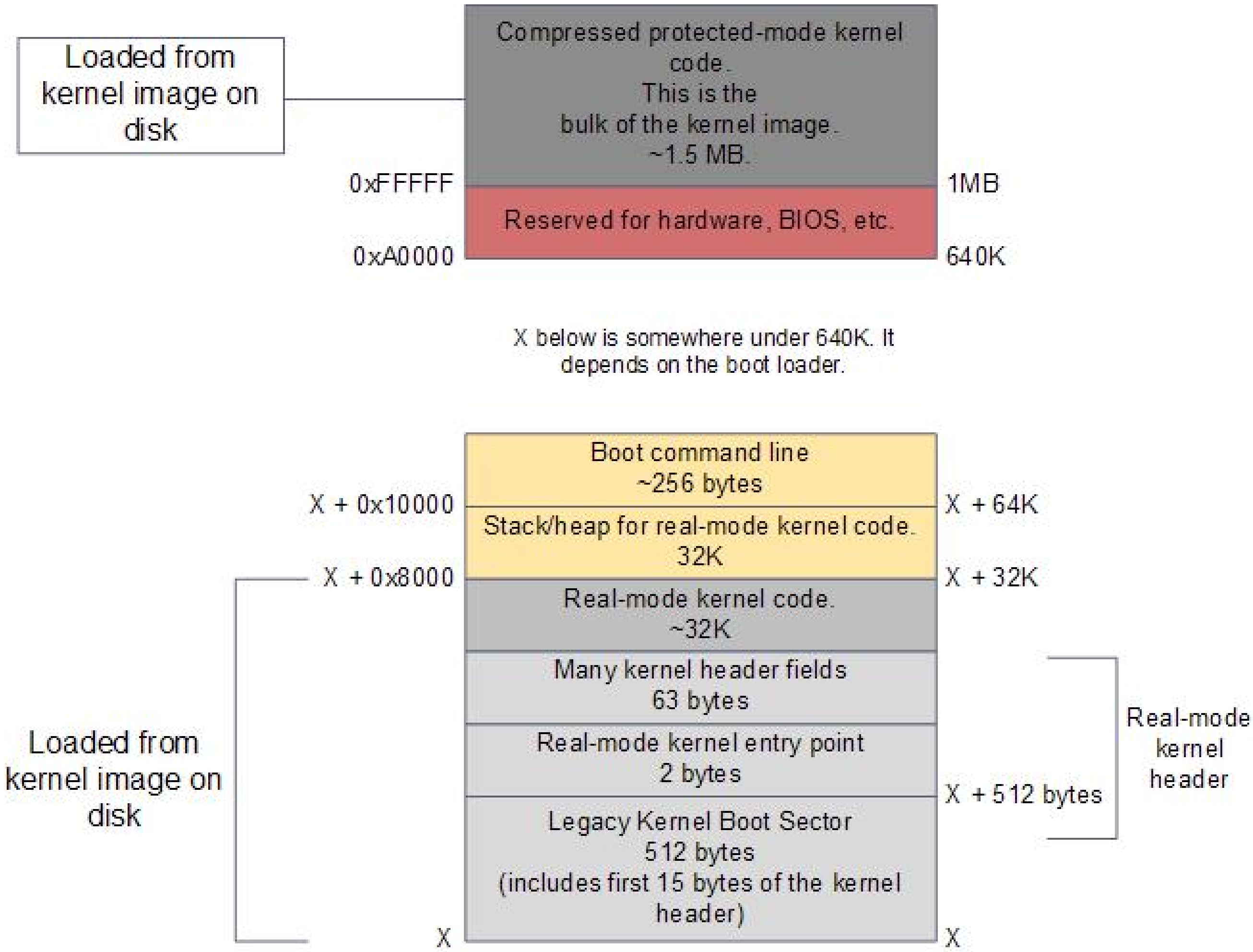
Boot Sequence

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, ...)

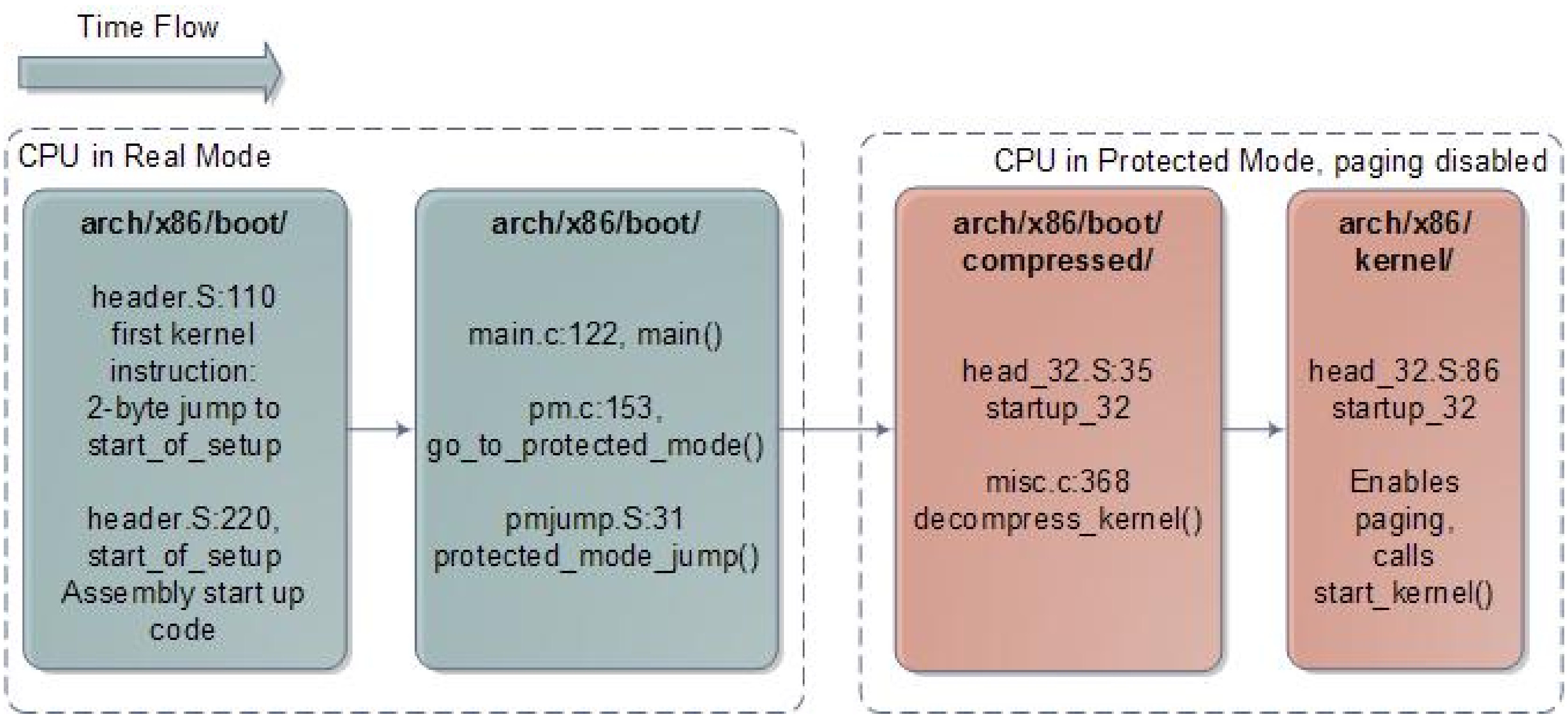
Initial Life of the Linux Kernel

- The Second stage bootloader (or the UEFI bootloader) loads the initial image of the kernel in memory
- This kernel image is way 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`

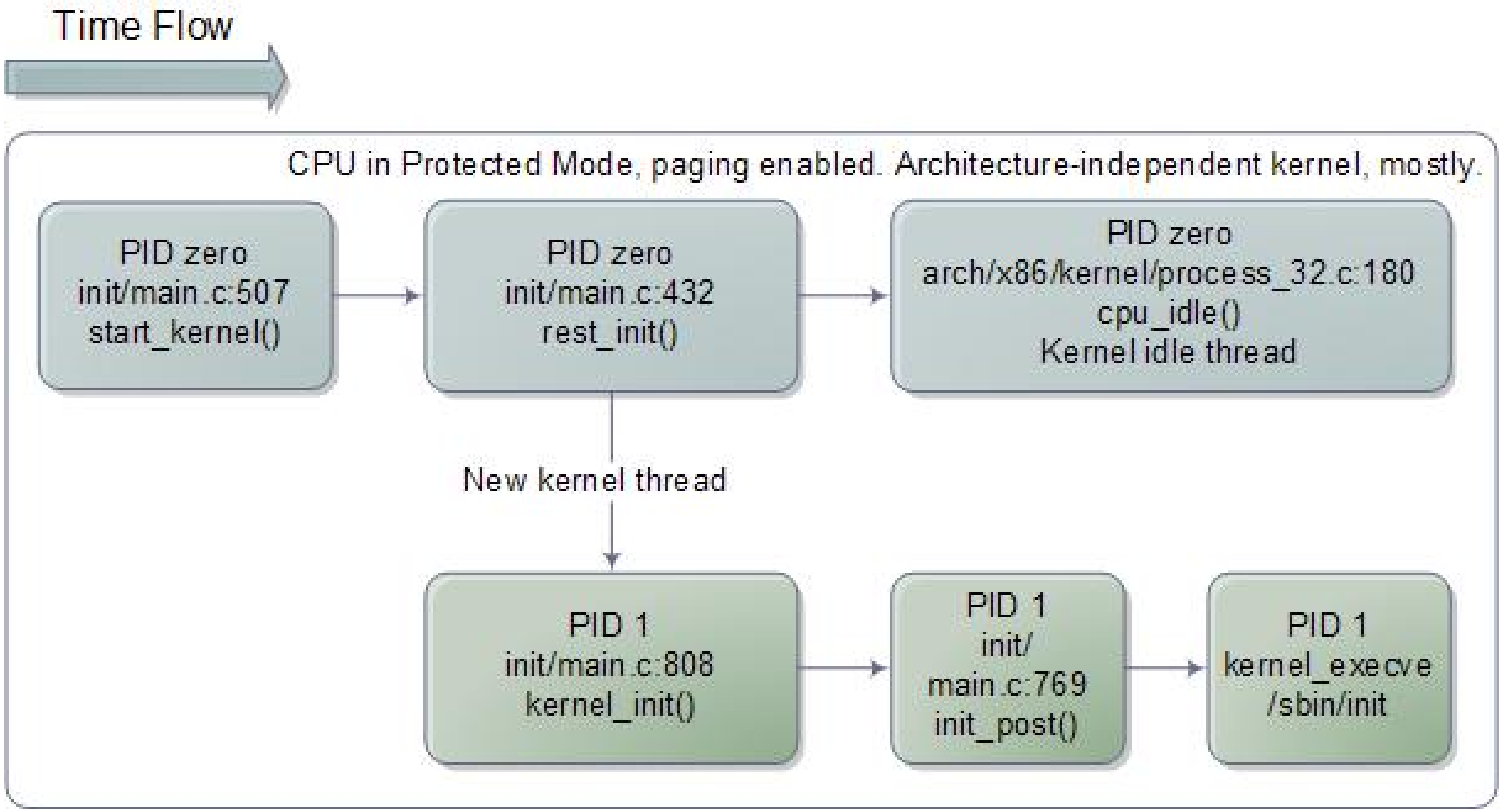
(Actual) Kernel entry point

- 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!

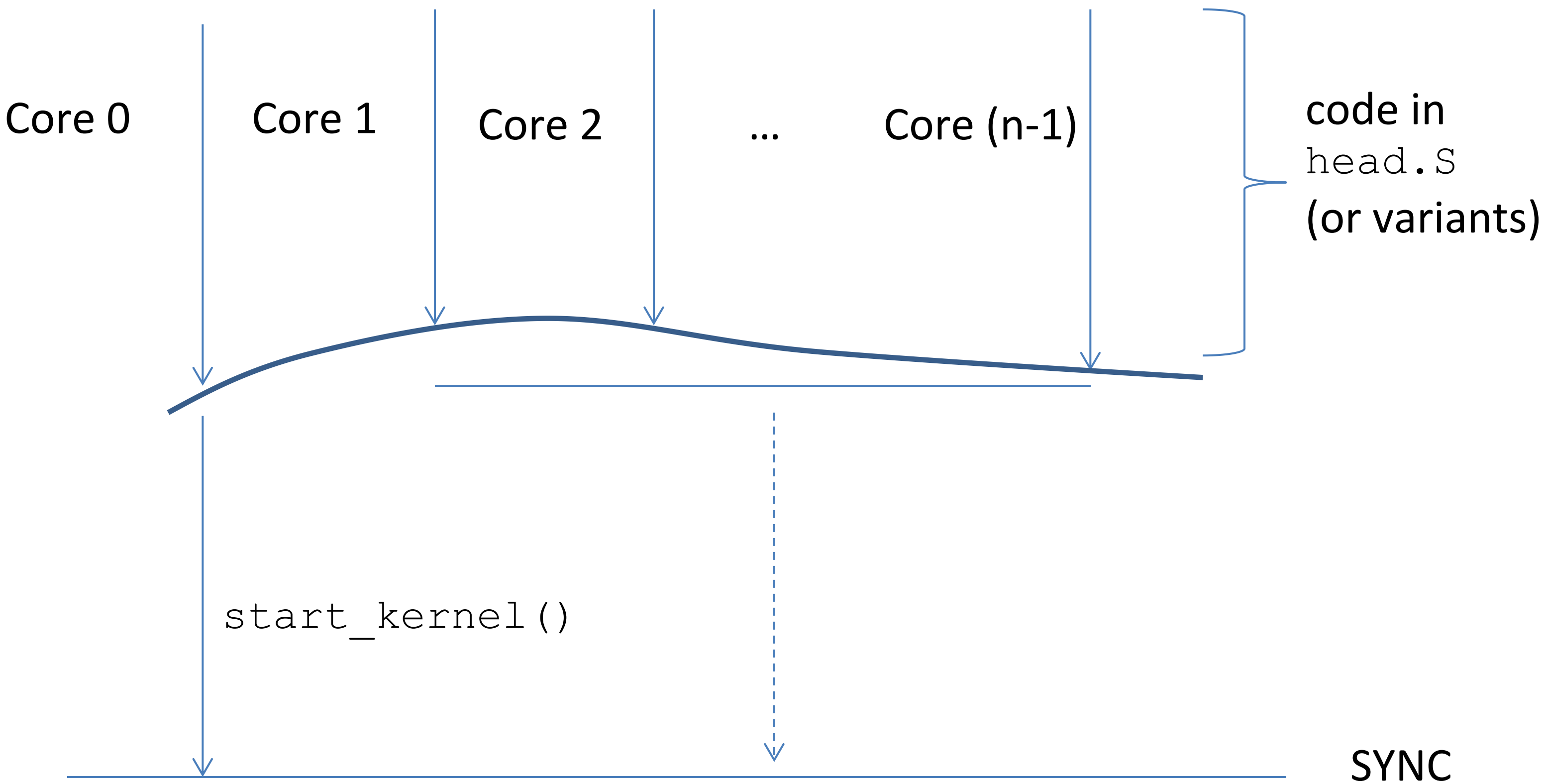
`startup_32()` (second version)

- 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
- "+": 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");
}
```

Using Inline Assebly in a C program

EXAMPLE SESSION

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)

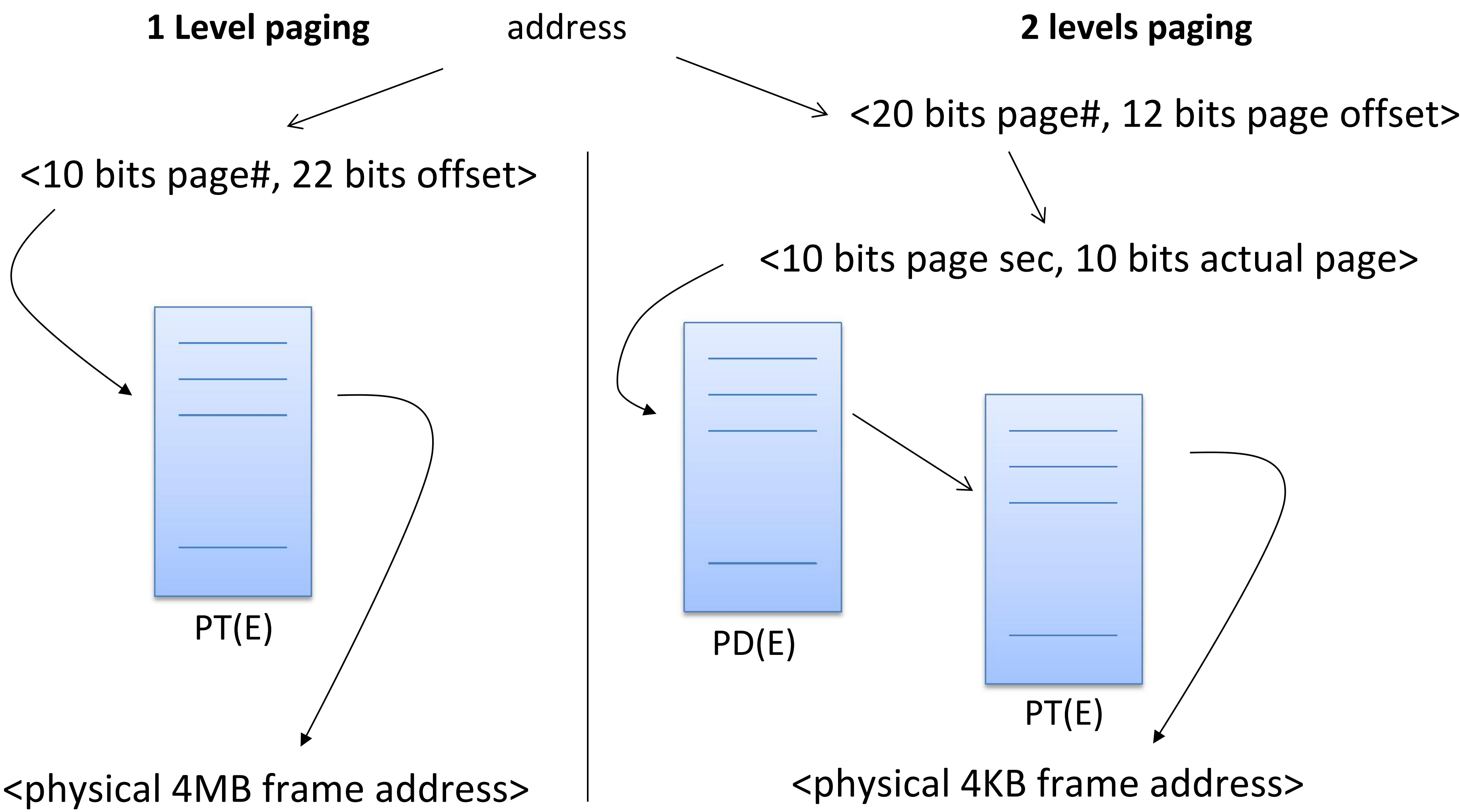
Some facts about memory

- 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

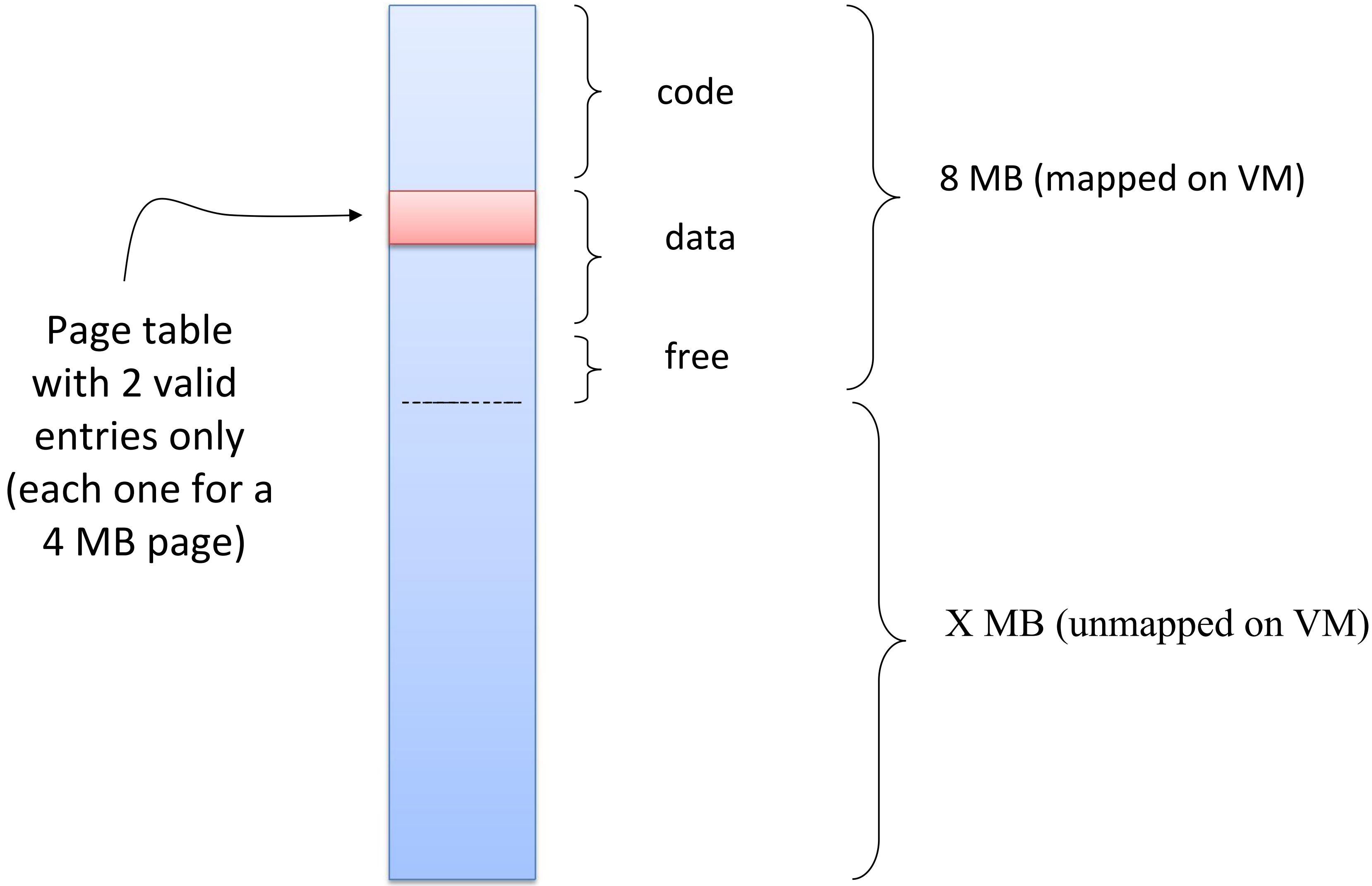
Enabling Paging

```
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)



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 (since core maps and free lists are not at steady state)
6. We need to find a way to describe the physical memory
7. We're not dealing with userspace memory yet!

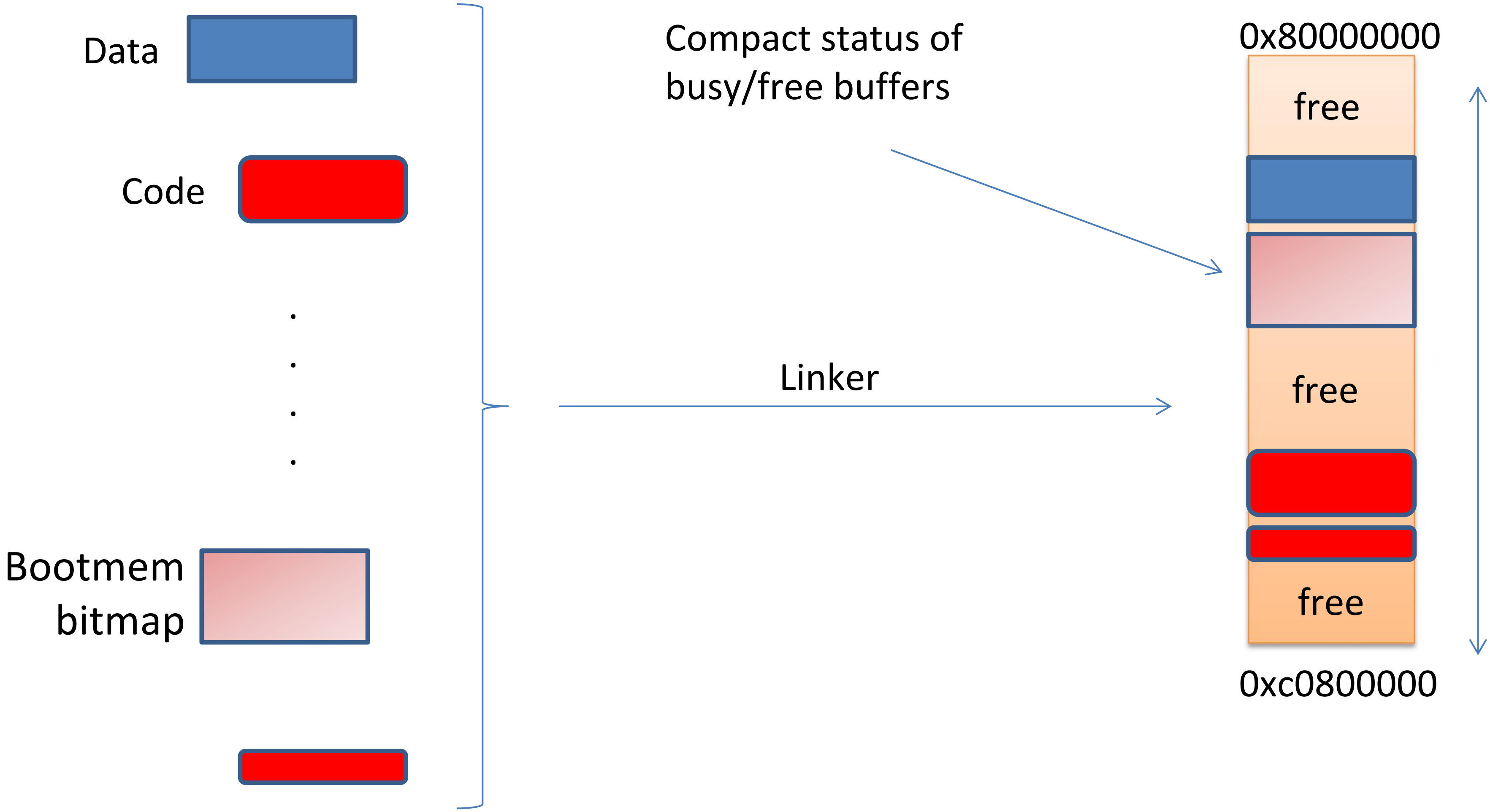
Kernel-Level MM Data Structures

- 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 (more on this later)
- Free list of physical memory frames, for any NUMA node

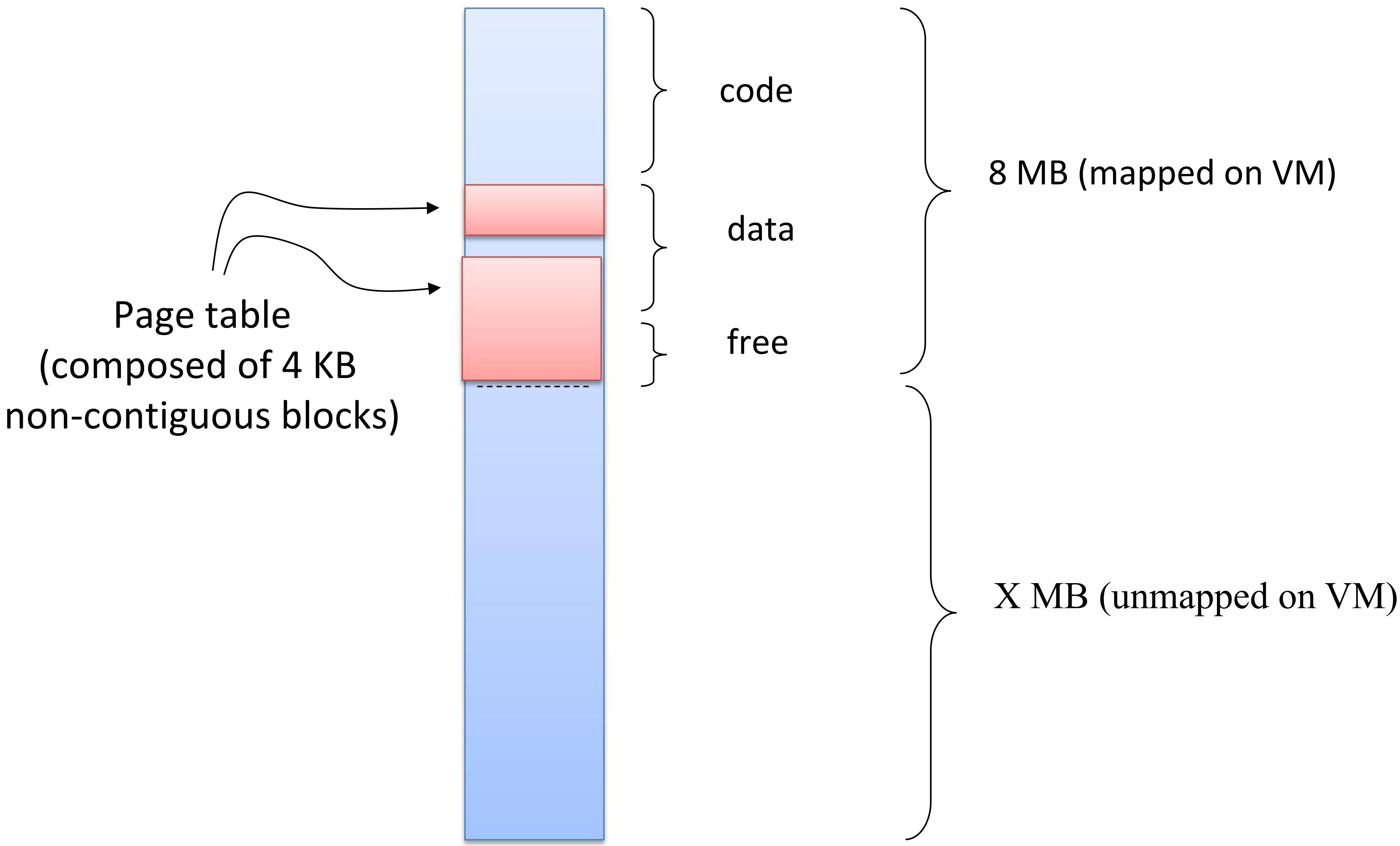
Bootmem

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

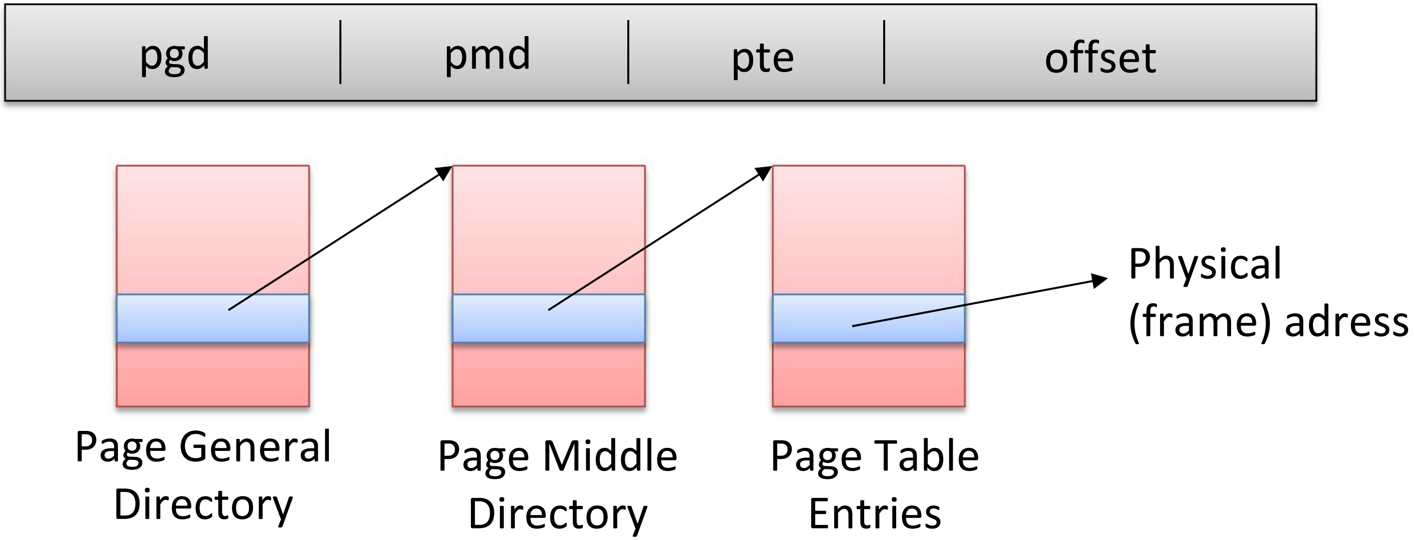


Collocation of PT in Physical Memory



How Linux handles paging

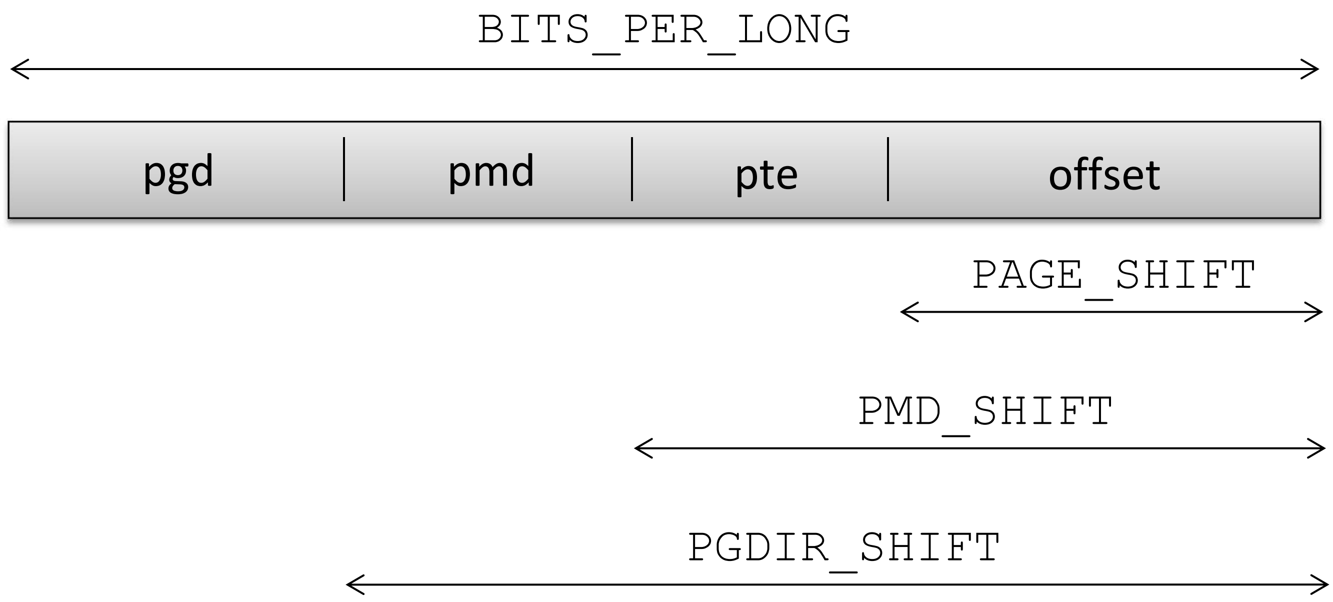
- Linux on x86 has 3 indirection levels:



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

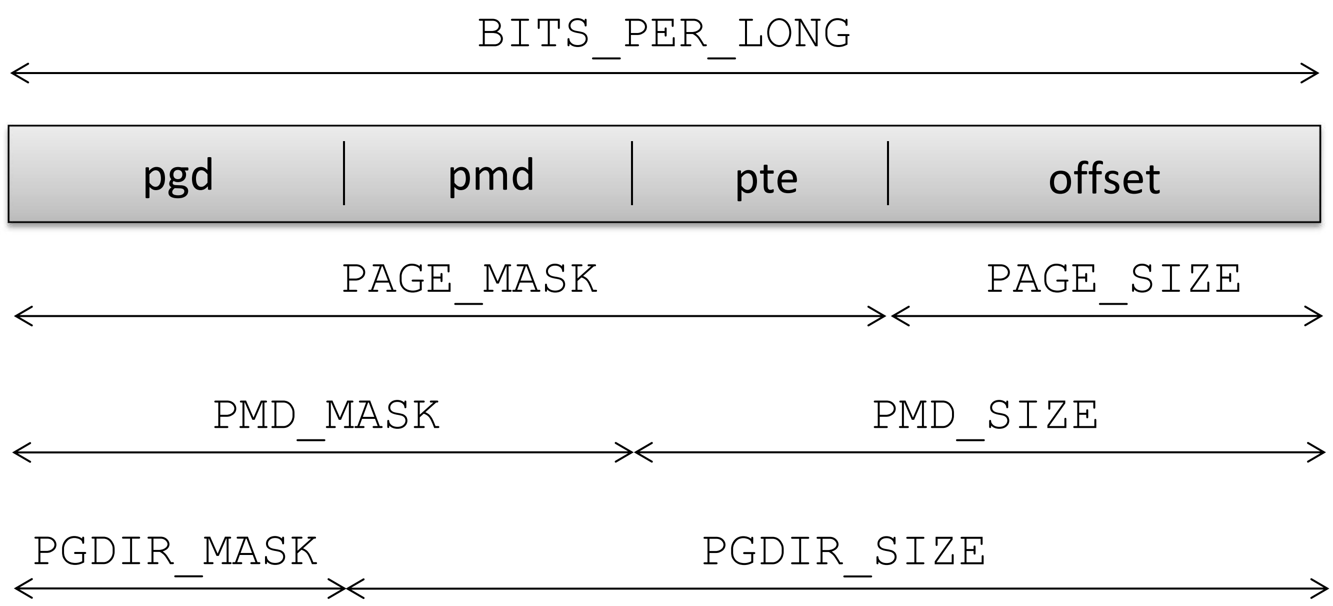
Splitting the address

- `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
#define PTRS_PER_PTE    1024
```

without PAE

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)
```

Page Types

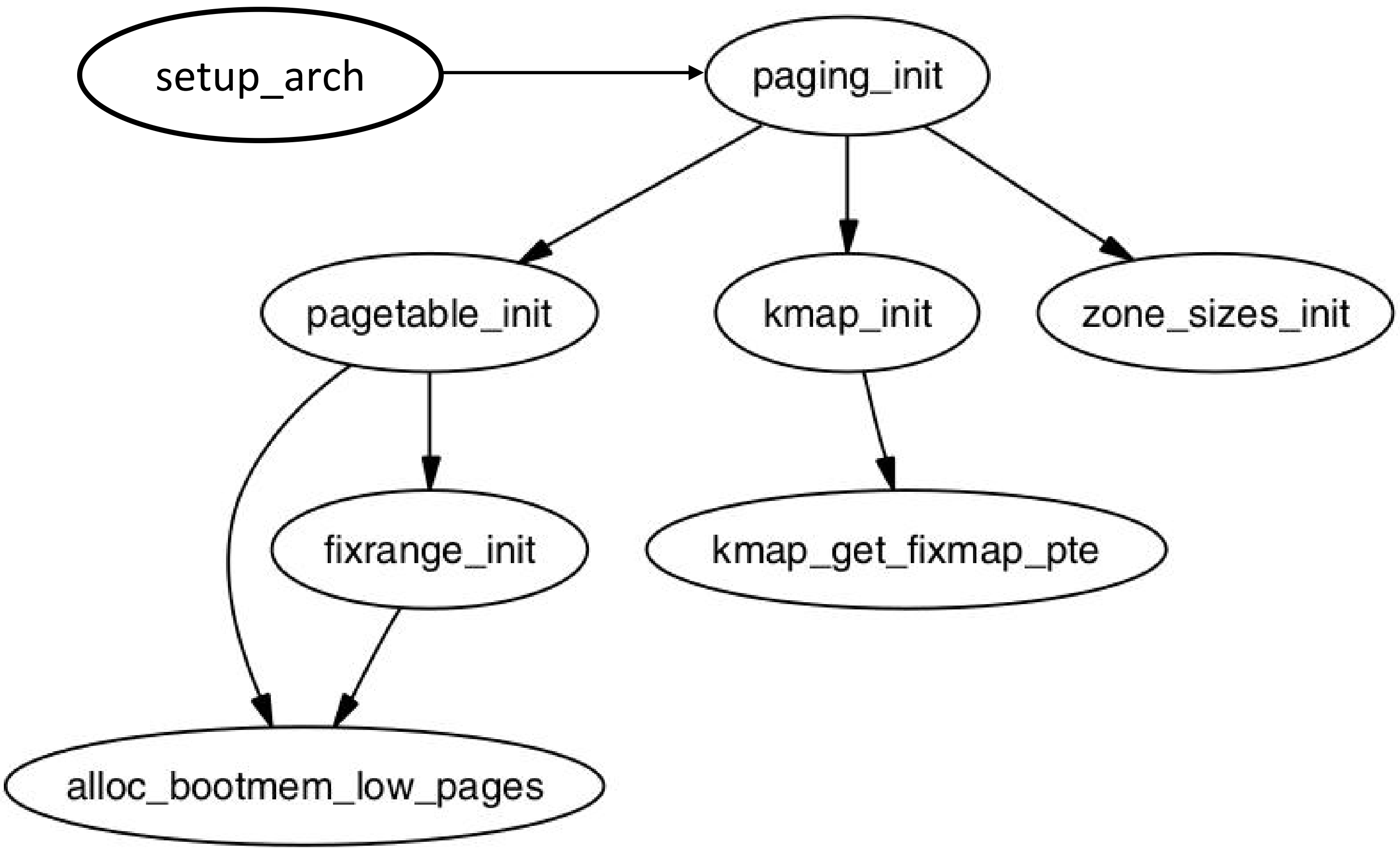
- `#define PAGE_SHARED __pgprot(_PAGE_PRESENT | _PAGE_RW | _PAGE_USER | _PAGE_ACCESSED)`
- `#define PAGE_READONLY __pgprot(_PAGE_PRESENT | _PAGE_USER | _PAGE_ACCESSED)`
- `#define __PAGE_KERNEL (_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY | _PAGE_ACCESSED)`
- `#define __PAGE_KERNEL_NOCACHE (_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY | _PAGE_PCD | _PAGE_ACCESSED)`
- `#define __PAGE_KERNEL_RO (_PAGE_PRESENT | _PAGE_DIRTY | _PAGE_ACCESSED)`

- Note that `_pgprot` expands to a cast to `pgprot_t` (still weak typing)

How to detect page size

```
#include <kernel.h>

#define MASK 1<<7

unsigned long addr = 3<<30; // Reference to the kernel boundary
                            // This is 0xC000 0000

asmlinkage int sys_page_size(void){
        if(swapper_pg_dir[(int)((unsigned long)addr>>22)] & MASK)
                return 4<<20;
        return 4<<10;
}
```

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();
```

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)
```

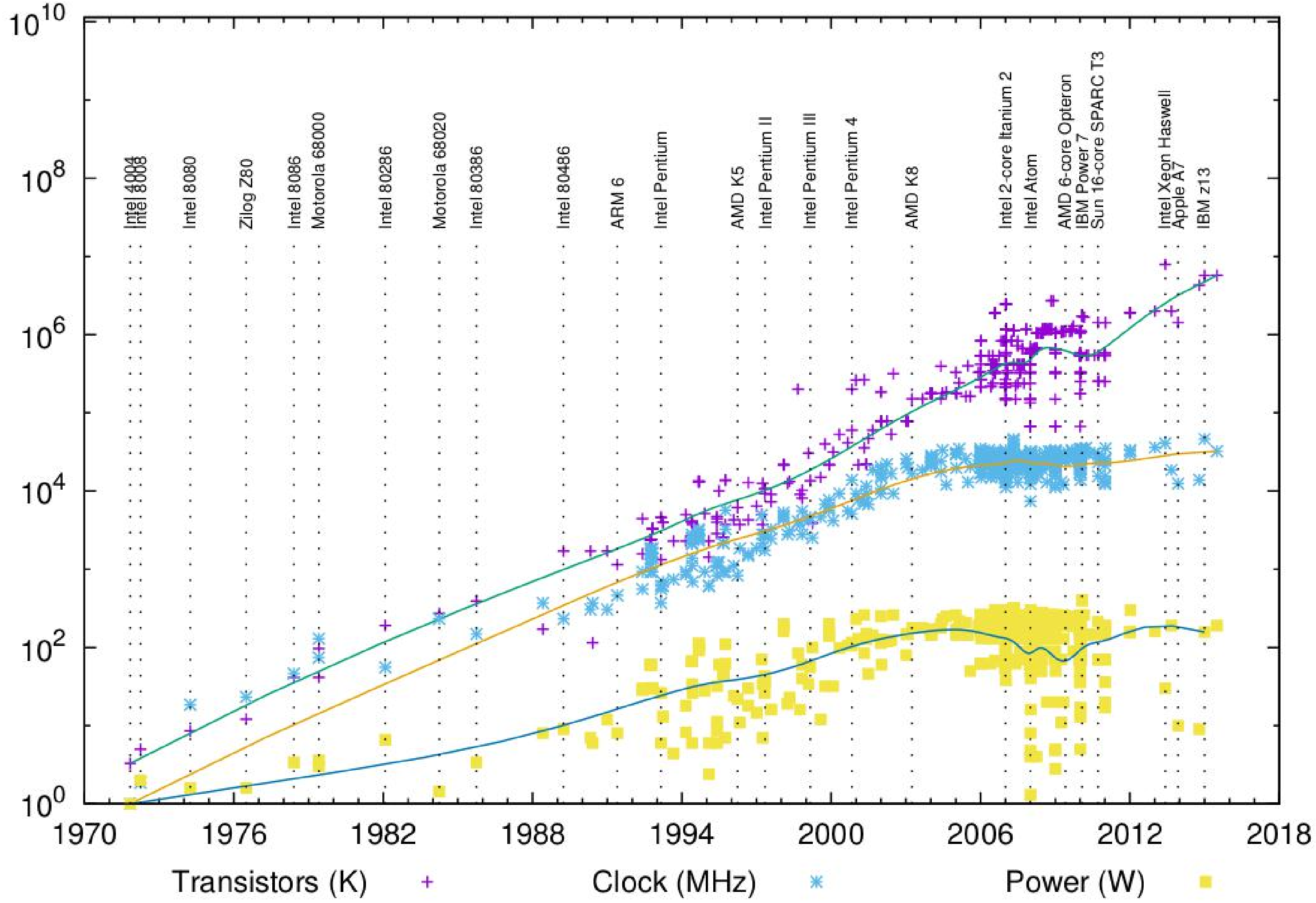
- Used when the page tables are being torn down and free'd
- Some platforms cache the lowest level of the page table, which needs to be flushed when the pages are being deleted (e.g. Sparc64)
- This is called when a region is being unmapped and the page directory entries are being reclaimed

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*

Modern Organization of RAM



Modern Organization of RAM

- The core count continuously increases
- It is becoming difficult to build architectures with a flat-latency memory access (historically referred to as UMA)
- Current machines are typically NUMA
- Each core has closer and farther RAM banks
- Each memory bank is associated with a NUMA node
- Modern operating systems are designed to handle NUMA machines (hence UMA as a special case)

Information on NUMA by the Kernel

- A very simple way to look into NUMA configuration is the `numactl` command
- It allows to discover:
  - How many NUMA nodes are available
  - What are the nodes close/far to/from any core
  - What is the distance from the cores of the nodes

- Go to the DCHPC classess for more on this!

Nodes Organization

- A node is organized in a `struct pglist_data` (even in the case of UMA) typedef'd to `pg_data_t`
- Every node in the system is kept on a NULL-terminated list called `pgdat_list`
- Each node is linked to the next with the field `pg_data_t→node_next`
  - In UMA systems, only one static `pg_data_t` structure called `contig_page_data` is used (defined at `mm/numa.c`)

Nodes Organization

- From Linux 2.6.16 to 2.6.17 much of the codebase of this portion of the kernel has been rewritten
- Introduction of macros to iterate over node data (most in `include/linux/mmzone.h`) such as:
  - `for_each_online_pgdat()`
  - `first_online_pgdat()`
  - `next_online_pgdat(pgdat)`
- Global `pgdat_list` has since then been removed
- Macros rely on the global `struct pglist_data *node_data[];`

pg_data_t

- Defined in `include/linux/mmzone.h`

```
typedef struct pglist_data {
        zone_t node_zones[MAX_NR_ZONES];
        zonelist_t node_zonelists[GFP_ZONEMASK+1];
        int nr_zones;
        struct page *node_mem_map;
        unsigned long *valid_addr_bitmap;
        struct bootmem_data *bdata;
        unsigned long node_start_paddr;
        unsigned long node_start_mapnr;
        unsigned long node_size;
        int node_id;
        struct pglist_data *node_next;
} pg_data_t;
```

Zones

- Nodes are divided into zones:

```
#define ZONE_DMA        0
#define ZONE_NORMAL     1
#define ZONE_HIGHMEM    2
#define MAX_NR_ZONES    3
```

- They target specific physical memory areas:
  - `ZONE_DMA`: < 16 MB
  - `ZONE_NORMAL`: 16-896 MB
  - `ZONE_HIGHMEM`: > 896 MB

Limited in size and high contention. Linux also has the notion of *high memory*

Zones Initialization

- zones are initialized after the kernel page tables have been fully set up by paging init()
- The goal is to determine what parameters to send to:
  - `free_area_init()` for UMA machines
  - `free_area_init_node()` for NUMA machines
- The initialization grounds on PFNs
- max PFN is read from BIOS e820 table

e820 dump in dmesg

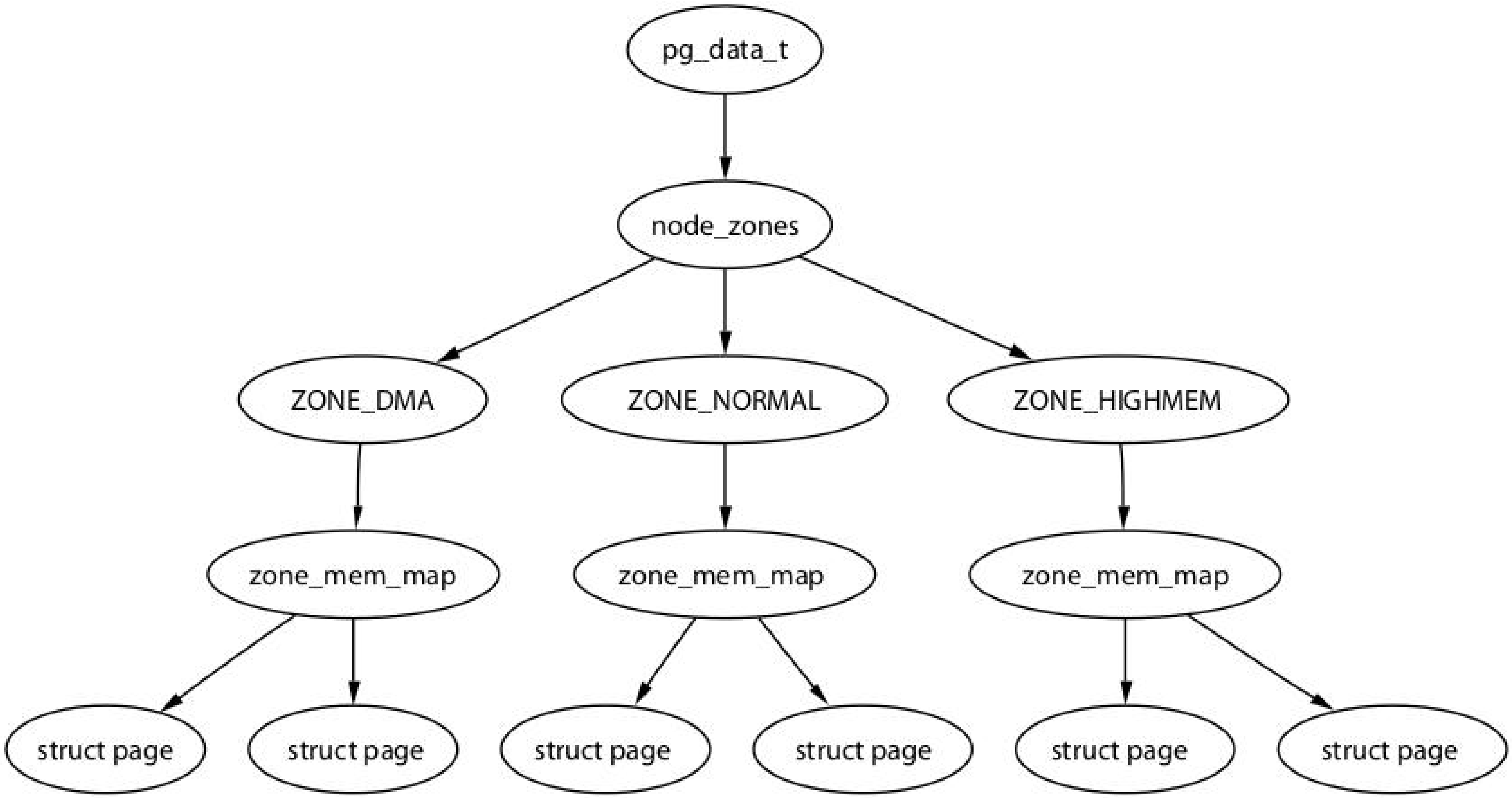
```
[0.000000] e820: BIOS-provided physical RAM map:
[0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000007dc08bff] usable
[0.000000] BIOS-e820: [mem 0x000000007dc08c00-0x000000007dc5cbff] ACPI NVS
[0.000000] BIOS-e820: [mem 0x000000007dc5cc00-0x000000007dc5ebff] ACPI data
[0.000000] BIOS-e820: [mem 0x000000007dc5ec00-0x000000007fffffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000e0000000-0x00000000efffffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000fed003ff] reserved
[0.000000] BIOS-e820: [mem 0x00000000fed20000-0x00000000fed9ffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000fee00000-0x00000000feefffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000ffb00000-0x00000000ffffffff] reserved
```

zone_t

```
typedef struct zone_struct {
        spinlock_t              lock;
        unsigned                long   free_pages;
        zone_watermarks_t       watermarks[MAX_NR_ZONES];
        unsigned long           need_balance;
        unsigned long       nr_active_pages,nr_inactive_pages;
        unsigned long           nr_cache_pages;
        free_area_t             free_area[MAX_ORDER];
        wait_queue_head_t       * wait_table;
        unsigned long           wait_table_size;
        unsigned long           wait_table_shift;
        struct pglist_data      *zone_pgdat;
        struct page             *zone_mem_map;
        unsigned long           zone_start_paddr;
        unsigned long           zone_start_mapnr;
        char                    *name;
        unsigned long           size;
        unsigned long           realsize;
} zone_t;
```

Currently 11

Nodes, Zones and Pages Relations



Core Map

- It is an array of `mem_map_t` structures defined in `include/linux/mm.h` and kept in `ZONE_NORMAL`

```
typedef struct page {
    struct list_head list;              /* ->mapping has some page lists. */
    struct address_space *mapping;      /* The inode (or ...) we belong to. */
    unsigned long index;                /* Our offset within mapping. */
    struct page *next_hash;             /* Next page sharing our hash bucket in
                                           the pagecache hash table. */
    atomic_t count;                     /* Usage count, see below. */
    unsigned long flags;                /* atomic flags, some possibly
                                           updated asynchronously */
    struct list_head lru;               /* Pageout list, eg. active_list;
                                           protected by pagemap_lru_lock !! */
    struct page **pprev_hash; /* Complement to *next_hash. */
    struct buffer_head * buffers;       /* Buffer maps us to a disk block. */

    #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
    void *virtual;                      /* Kernel virtual address (NULL if
                                           not kmapped, ie. highmem) */
    #endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
} mem_map_t;
```

Core Map Members

- Struct members are used to keep track of the interactions between MM and other kernel sub-systems
- `struct list_head list`: used to organize the frames into free lists
- `atomic_t count`: counts the virtual references mapped onto the frame
- `unsigned long flags`: status bits for the frame

```
#define PG_locked       0
#define PG_referenced 2
#define PG_uptodate    3
#define PG_dirty       4
#define PG_lru         6
#define PG_reserved   14
```

How to manage `flags`

Bit Name	Set	Test	Clear
PG_active	SetPageActive()	PageActive()	ClearPageActive()
PG_arch_1	None	None	None
PG_checked	SetPageChecked()	PageChecked()	None
PG_dirty	SetPageDirty()	PageDirty()	ClearPageDirty()
PG_error	SetPageError()	PageError()	ClearPageError()
PG_highmem	None	PageHighMem()	None
PG_launder	SetPageLaunder()	PageLaunder()	ClearPageLaunder()
PG_locked	LockPage()	PageLocked()	UnlockPage()
PG_lru	TestSetPageLRU()	PageLRU()	TestClearPageLRU()
PG_referenced	SetPageReferenced()	PageReferenced()	ClearPageReferenced()
PG_reserved	SetPageReserved()	PageReserved()	ClearPageReserved()
PG_skip	None	None	None
PG_slab	PageSetSlab()	PageSlab()	PageClearSlab()
PG_unused	None	None	None
PG_uptodate	SetPageUptodate()	PageUptodate()	ClearPageUptodate()

Core Map on UMA

- Initially we only have the core map pointer
- This is `mem_map` and is declared in `mm/memory.c`
- Pointer initialization and corresponding memory allocation occur within `free_area_init()`
- After initializing, each entry will keep the value 0 within the `count` field and the value 1 into the `PG_reserved` flag within the `flags` field
- Hence no virtual reference exists for that frame and the frame is reserved
- Frame un-reserving will take place later via the function `mem_init()` in `arch/i386/mm/init.c` (by resetting the bit `PG_reserved`)

Core Map on NUMA

- There is not a global mem_map array
- Every node keeps its own map in its own memory
- This map is referenced by `pg_data_t→node_mem_map`
- The rest of the organization of the map does not change

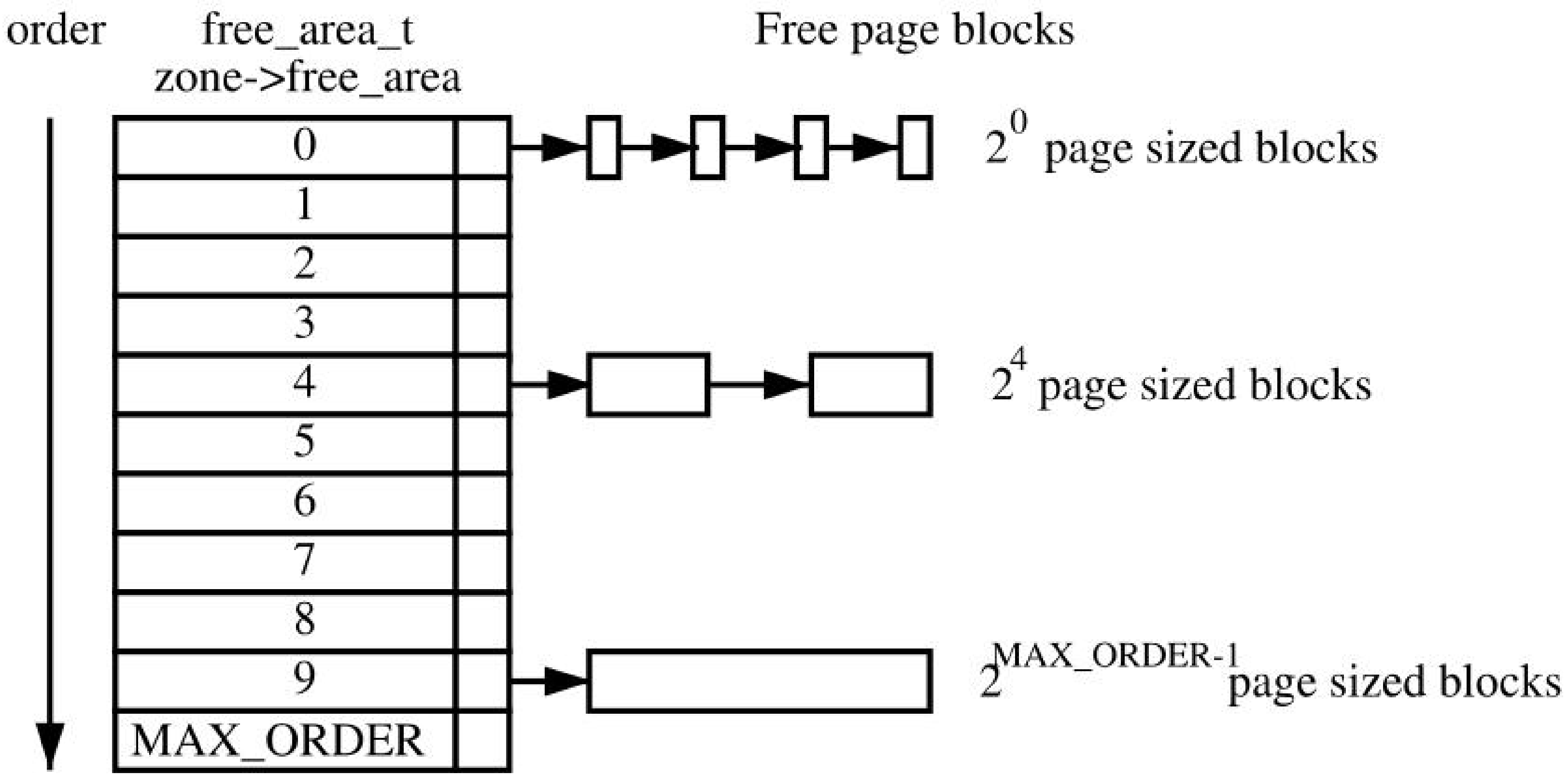
Buddy System: Frame Allocator

- By Knowlton (1965) and Knuth (1968)
- It has been experimentally shown to be quite fast

- Based on two main data structures:

```
typedef struct free_area_struct {
    struct list_head list;
    unsigned int *map;
} free_area_t

struct list_head {
    struct list_head *next, *prev;
}
```

`free_area_t` organization



Bitmap `*map` semantic

- Linux saves memory by using one bit for a pair of buddies
- It's a "fragmentation" bit
- Each time a buddy is allocated or free'd, the bit representing the pair is toggled
  - 0: if the pages are both free or allocated
  - 1: only one buddy is in use

Index in the global mem_map array

```
#define MARK_USED(index, order, area) \
__change_bit((index) >> (1+(order)), (area)->map)
```

High Memory

- When the size of physical memory approaches/ exceeds the maximum size of virtual memory, it is impossible for the kernel to keep all of the available physical memory mapped
- "Highmem" is the memory not covered by a permanent mapping
- The Kernel has an API to allow "temporary mappings"

High Memory

- `vmap()`: used to make a long-duration mapping of multiple physical pages
- `kmap()`: it permits a short-duration mapping of a single page.
  - It needs global synchronization, but is amortized somewhat.
- `kmap_atomic()`: This permits a very short duration mapping of a single page.
  - It is restricted to the CPU that issued it
  - the issuing task is required to stay on that CPU until it has finished

- In general: nowadays, it *really* makes sense to use 64-bit systems!

High Memory Deallocation

- Kernel maintains an array of counters:

```
static int pkmap_count[ LAST_PKMAP ];
```

- One counter for each 'high memory' page
- Counter values are 0, 1, or more than 1:
  - =0: page is not mapped
  - =1: page not mapped now, but used to be
  - =n >1: page was mapped (n-1) times

kunmap()

- `kunmap(page)` decrements the associated reference counter
- When the counter is 1, mapping is not needed anymore
- But CPU still has ‘cached’ that mapping
- So the mapping must be ‘invalidated’
- With multiple CPUs, all of them must do it
  - `__flush_tlb_all()`

Finalizing Memory Initialization

- The finalization of memory management init is done via `mem_init()` which destroys the bootmem allocator
- This function will release the frames, by resetting the `PG_RESERVED` bit
- For each free'd frame, the function `__free_page()` is invoked
  - This gives all the pages in ZONE_NORMAL to the buddy allocator
- At this point the reference `count` within the corresponding entry gets set to 1 since the kernel maps that frame anyway within its page table

Finalizing Memory Initialization

```
static unsigned long __init
free_all_bootmem_core(pg_data_t *pgdat) {
        ……………
        // Loop through all pages in the current node
        for (i = 0; i < idx; i++, page++) {
                if (!test_bit(i, bdata->node_bootmem_map)) {
                        count++;
                        ClearPageReserved(page);
                        // Fake the buddy into thinking it's an
                        // actual free
                        set_page_count(page, 1);
                        __free_page(page);
                }
        }
        total += count;
        ……………
        return total;
}
```

Allocation Contexts

- Process context: allocation due to a system call
  - If it cannot be served: wait along the current execution trace
  - Priority-based approach
- Interrupt: allocation due to an interrupt handler
  - If it cannot be served: no actual waiting time
  - Priority independent schemes

- This approach is general to most Kernel subsystems

Basic Kernel Internal MM API

- At steady state, the MM subsystem exposes API to other kernel subsystems
- Prototypes in `#include <linux/malloc.h>`
- Basic API: page allocation
  - `unsigned long get_zeroed_page(int flags)`: take a frame from the free list, zero the content and return its virtual address
  - `unsigned long __get_free_page(int flags)`: take a frame from the free list and return its virtual address
  - `unsigned long __get_free_pages(int flags, unsigned long order)`: take a block of contiguous frames of given `order` from the free list

Basic Kernel Internal MM API

- Basic API: page allocation
  - `void free_page(unsigned long addr)`: put a frame back into the free list
  - `void free_pages(unsigned long addr, unsigned long order)`: put a block of frames of given order back into the free list
- Warning: passing a wrong `addr` or `order` might corrupt the Kernel!

Basic Kernel Internal MM API

- `flags`: used to specify the allocation context
  - `GFP_ATOMIC`: interrupt context. The call cannot lead to sleep
  - `GFP_USER`: Used to allocate memory for userspace. The call can lead to sleep
  - `GFP_BUFFER`: Used to allocate a buffer. The call can lead to sleep
  - `GFP_KERNEL`: Used to allocate Kernel memory. The call can lead to sleep

NUMA Allocation

- On NUMA systems, we have multiple nodes
- UMA systems eventually invoke NUMA API, but the system is configured to have a single node
- Core memory allocation API:
  - `struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);`
  - `__get_free_pages()` calls `alloc_pages_node()` specifying a *NUMA policy*

NUMA Policies

- NUMA policies determine what NUMA node is involved in a memory operation
- Since Kernel 2.6.18, userspace can tell the Kernel what policy to use:

```
#include <numaif.h>
int set_mempolicy(int mode, unsigned long
*nodemask, unsigned long maxnode);
```

- `mode` can be: `MPOL_DEFAULT`, `MPOL_BIND`, `MPOL_INTERLEAVE` or `MPOL_PREFERRED`

NUMA Policies

```
#include <numaif.h>
int mbind(void *addr, unsigned long len,
int mode, unsigned long *nodemask,
unsigned long maxnode, unsigned flags);
```

Sets the NUMA memory policy, which consists of a policy mode and zero or more nodes, for the memory range starting with *addr* and continuing for *len* bytes. The memory policy defines from which node memory is allocated.

Moving Pages Around

```
#include <numaif.h>
long move_pages(int pid, unsigned long
count, void **pages, const int *nodes,
int *status, int flags);
```

moves the specified *pages* of the process *pid* to the memory nodes specified by *nodes*. The result of the move is reflected in *status*. The *flags* indicate constraints on the pages to be moved.

Frequent Allocations/Deallocations

- Consider fixed-size data structures which are frequently allocated/released
- The buddy system here does not scale
  - This is a classical case of frequent logical contention
  - The Buddy System on each NUMA node is protected by a spinlock
  - The internal fragmentation might rise too much
- There is a dedicated allocator for fixed-size data structures (referred to as *slabs*)

Classical Examples

- Allocation/release of page tables, at any level, is very frequent
- It is mandatory not to lose time in this operation
- Quicklists are used to this purpose
- For paging we have:
  - `pgd_alloc()`, `pmd_alloc()` and `pte_alloc()`
  - `pgd_free()`, `pmd_free()` and `pte_free()`

Fast Allocation

- There are several fast allocators in the Kernel
- For paging, there are the *quicklists*
- For other buffers, there is the *slab allocator*
- There are three implementations of the slab allocator in Linux:
  - the SLAB: Implemented around 1994
  - the SLUB: The Unqueued Slab Allocator, default since Kernel 2.6.23
  - the SLOB: Simple List of Blocks. If the SLAB is disabled at compile time, Linux reverts to this

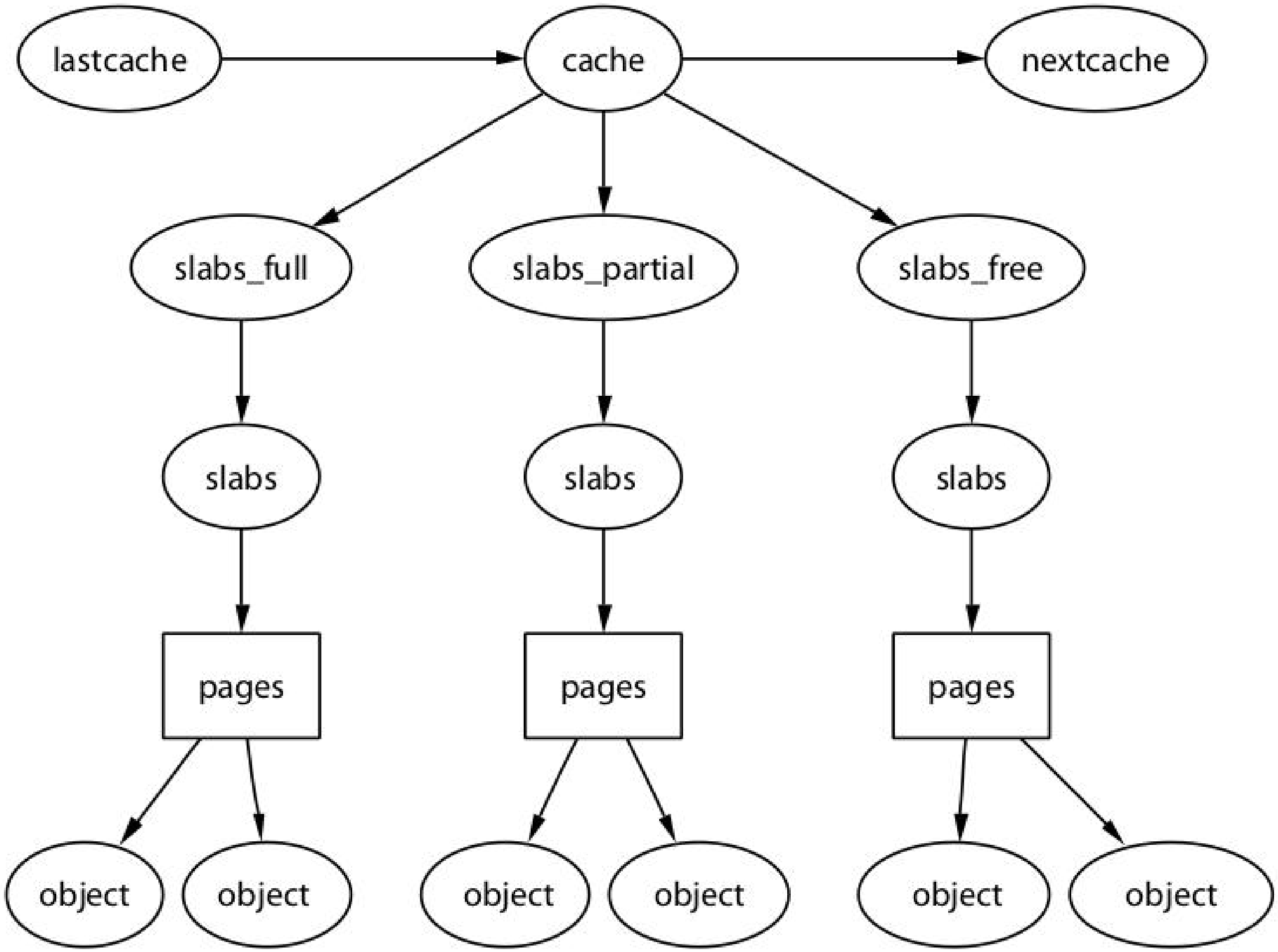
Quicklist

- Defined in `include/linux/quicklist.h`
- They are implemented as a list of per-core page lists
- There is no need for synchronization
- If allocation fails, they revert to `__get_free_page()`

Quicklist Allocation

```
static inline void *quicklist_alloc(int nr, gfp_t flags, ...) {
	struct quicklist *q;
	void **p = NULL;

	q = &get_cpu_var(quicklist)[nr];
	p = q->page;
	if (likely(p)) {
		q->page = p[0];
		p[0] = NULL;
		q->nr_pages--;
	}
	put_cpu_var(quicklist);
	if (likely(p))
		return p;

	p = (void *)__get_free_page(flags | __GFP_ZERO);
	return p;
}
```

The SLAB Allocator



SLAB Interfaces

- Prototypes are in `#include <linux/malloc.h>`
- `void *kmalloc(size_t size, int flags)`: allocation of contiguous memory (it returns the virtual address)
- `void kfree(void *obj)`: frees memory allocated via `kmalloc()`
- `void *kmalloc_node(size_t size, int flags, int node)`: NUMA-aware allocation

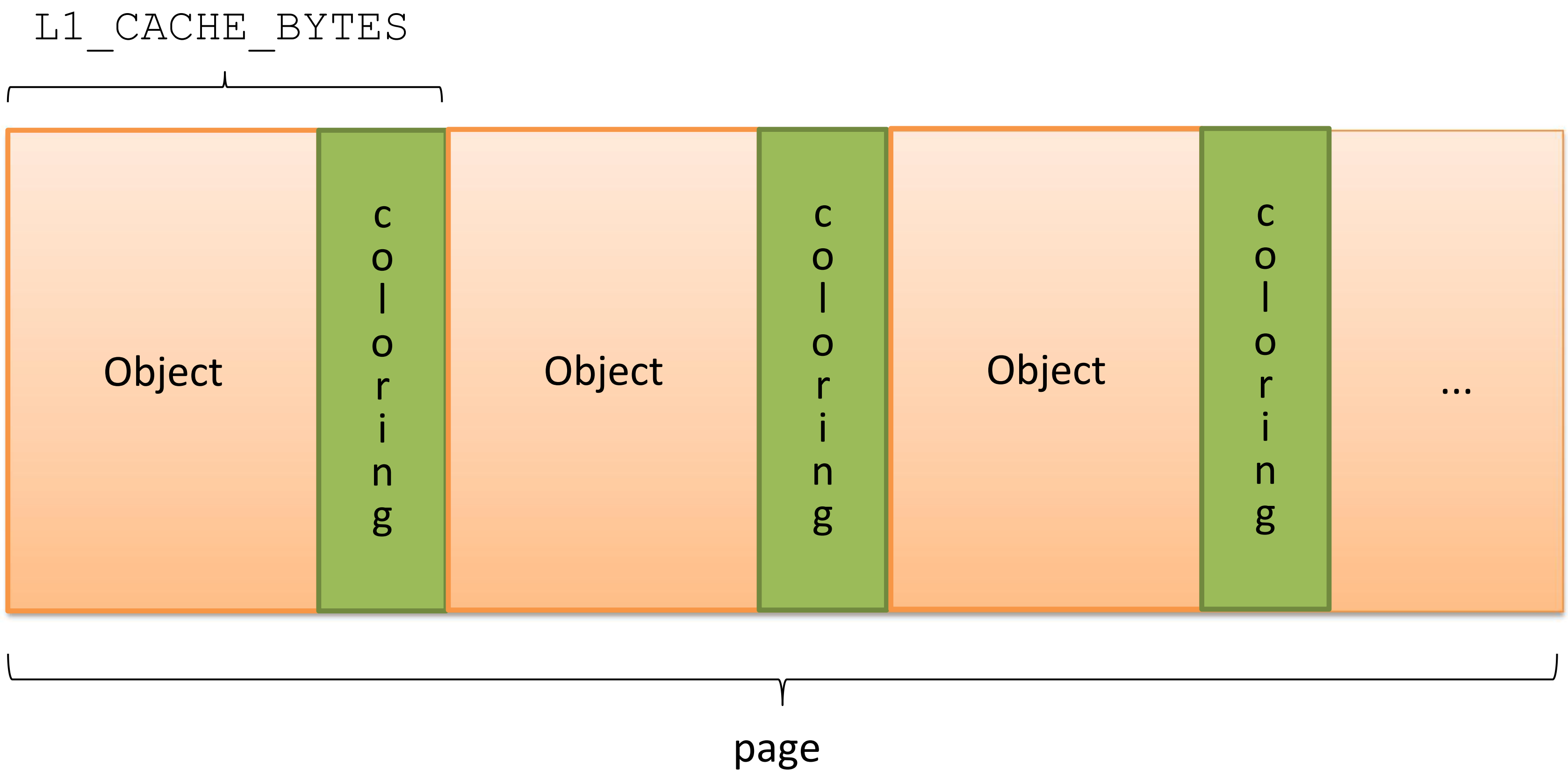
Available Caches (up to 3.9.11)

```
struct cache_sizes {
	size_t                  cs_size;
	struct kmem_cache       *cs_cachep;
#ifdef CONFIG_ZONE_DMA
	struct kmem_cache       *cs_dmacachep;
#endif
}

static cache_sizes_t cache_sizes[] = {
	{32,        NULL,       NULL},
	{64,        NULL,       NULL},
	{128,       NULL,       NULL}
	...
	{65536,     NULL,       NULL},
	{131072,    NULL,       NULL},
}
```

Available Caches (since 3.10)

```
struct kmem_cache_node {
        spinlock_t list_lock;

#ifdef CONFIG_SLAB
        struct list_head slabs_partial; /* partial list first, better asm code */
        struct list_head slabs_full;
        struct list_head slabs_free;
        unsigned long free_objects;
        unsigned int free_limit;
        unsigned int colour_next;       /* Per-node cache coloring */
        struct array_cache *shared;     /* shared per node */
        struct array_cache **alien;     /* on other nodes */
        unsigned long next_reap;        /* updated without locking */
        int free_touched;               /* updated without locking */
#endif
};
```

Slab Coloring



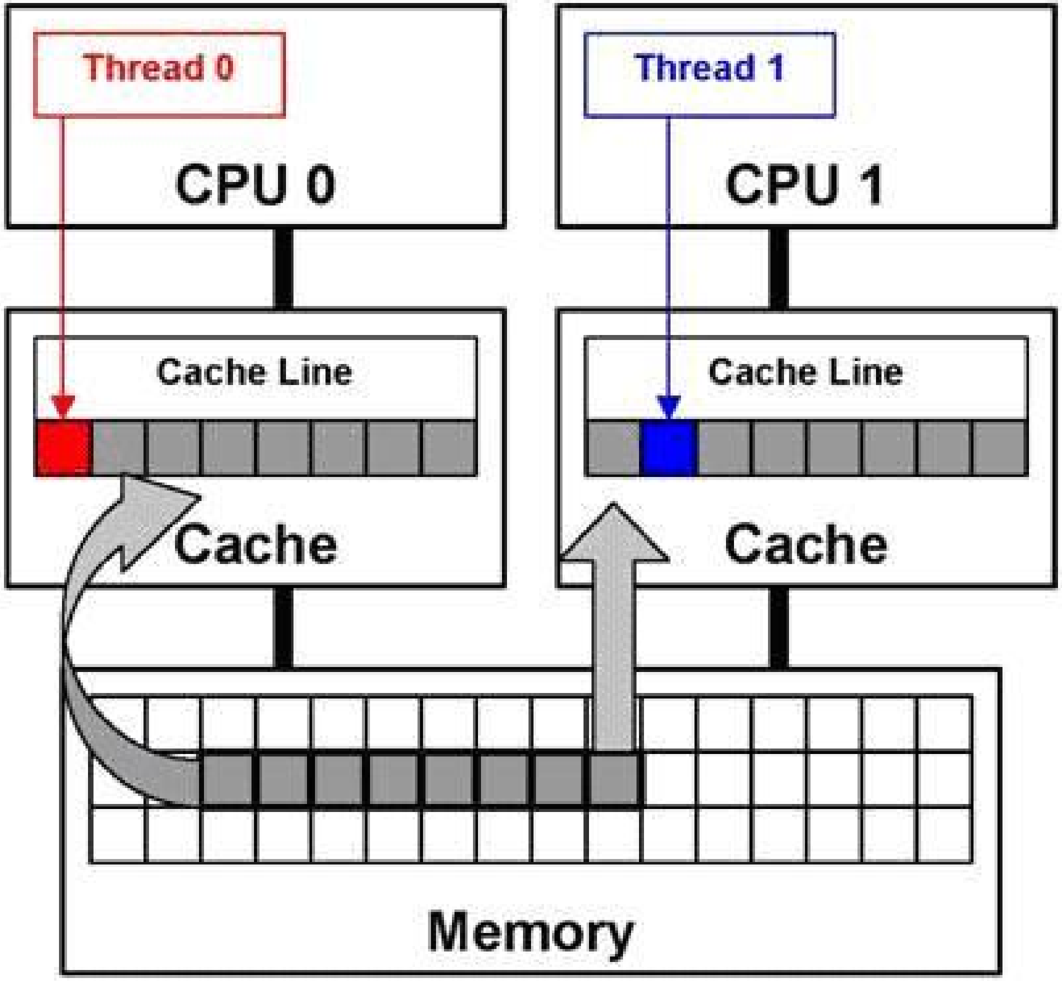
L1 data caches

- Cache lines are small (typically 32/64 bytes)
- `L1_CACHE_BYTES` is the configuration macro in Linux
- Independenlty of the mapping scheme, close addresses fall in the same line
- cache-aligned addressess fall in different lines
- We need to cope with *cache performance issues at the level of kernel programming* (typically not of explicit concern for user level programming)

Cache Performance Aspects

- *Common members* access issues
  - Most-used members in a data structure should be placed at its head to maximize cache hits
  - This should happen provided that the slab-allocation (kmalloc()) system gives cache-line aligned addresses for dynamically allocated memory chunks
- *Loosely related fields* should be placed sufficiently distant in the data structure so as to avoid performance penalties due to *false cache sharing*

The false cache sharing problem



Cache flush operations

- Cache flushes automation can be partial (similar to TLB)
- Need for explicit cache flush operations
- In some cases, the flush operation uses the physical address of the cached data to support flushing ("strict caching systems", e.g. HyperSparc)
- Hence, TLB flushes should always be placed after the corresponding data cache flush calls

Flushing Full MM	Flushing Range	Flushing Page
flush_cache_mm() Change all page tables flush_tlb_mm()	flush_cache_range() Change page table range flush_tlb_range()	flush_cache_page() Change single PTE flush_tlb_page()

Cache flush operations

- `void flush_cache_all(void)`
  - flushes the entire CPU cache system, which makes it the most severe flush operation to use
  - It is used when changes to the kernel page tables, which are global in nature, are to be performed

- `void flush_cache_mm(struct mm_struct *mm)`
  - flushes all entries related to the address space
  - On completion, no cache lines will be associated with `mm`

Cache flush operations

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

- This flushes lines related to a range of addresses
- Like its TLB equivalent, it is provided in case the architecture has an efficient way of flushing ranges instead of flushing each individual page

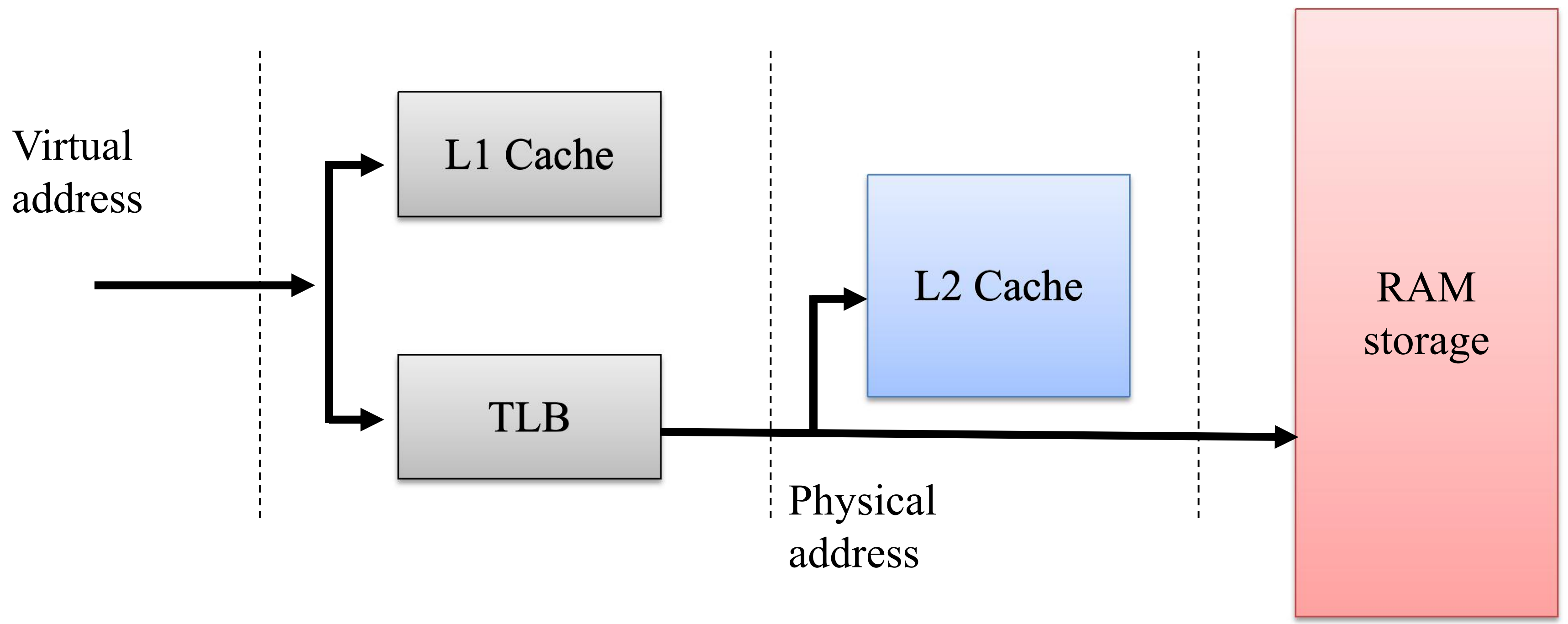
```
void flush_cache_page(struct vm_area_struct
*vma, unsigned long vmaddr)
```

- flushes a single-page-sized region
- `vma` is supplied because the `mm_struct` is easily accessible through `vma→vm_mm`
- Additionally, by testing for the `VM_EXEC` flag, the architecture knows if the region is executable for caches that separate the instructions and data caches

Racing inside the caching architecture

- What is better to manage caches? Virtual or physical address?
  - Virtual address are available as soon as addressing is resolved
  - Physical address require TLB translation
- With physical addresses we pay (in the hit case) the cache access cost twice
- In typical architectures, the optimal performance is achieved by having the L1 cache and the TLB racing to provide their outputs for subsequent use

Racing inside the caching architecture



x86 Caches

- On x86 architectures, caches are physically indexed and physically tagged (except for small L1 caches)
- Explicit cache flush operations are not required
- This is because a virtual address associated with any memory map is filtered by the MMU before real access to the memory hierarchy is performed

Virtual aliasing

- This is an anomaly occurring when the cache (at some level) is indexed via virtual addresses (e.g. Sparc64)
- The same RAM location can be associated with multiple virtual addresses
- Hence the RAM location can be mapped on multiple cache lines
- This leads to cache coherency issues
- Typical scenarios:
    - Shared memory in user space
    - Kernel/user page sharing

Solutions

- Hardware:
  - Arrange the cache in a way that only one virtual alias can be in the cache at any given time (works well for small size caches – e.g. L1)
- Software:
  - Map shared memory segments on conflicting cache lines
  - Flush the cache at context switches (again for cross-process coherency)
  - Flush the cache when mapping a page in the user address space section (this also works for kernel/user sharing of the mapped RAM address)

Cache flush API (examples)

- `void flush_dcache_page(struct page *page)`
  - Called when the kernel writes to or copies from a page-cache page because these are likely to be mapped by multiple processes

- `void flush_icache_range(unsigned long address, unsigned long endaddr)`
  - This is called when the kernel stores information in addresses that is likely to be executed (a kernel module has been loaded)

- `void flush_icache_page(struct vm_area_struct *vma, struct page *page)`
  - This is called when a page-cache page is about to be mapped. It is up to the architecture to use the `vma` flags to determine whether the I-Cache or D-Cache should be flushed

User-/Kernel-Level Data Movement

```
unsigned long copy_from_user(void *to, const void *from,
   unsigned long n)
```

Copies n bytes from the user address(from) to the kernel address space(to).

```
unsigned long copy_to_user(void *to, const void *from,
   unsigned long n)
```

Copies n bytes from the kernel address(from) to the user address space(to).

```
void get_user(void *to, void *from)
```

Copies an integer value from userspace (from) to kernel space (to).

```
void put_user(void *from, void *to)
```

Copies an integer value from kernel space (from) to userspace (to).

```
long strncpy_from_user(char *dst, const char *src, long count)
```

Copies a null terminated string of at most count bytes long from userspace (src) to kernel space (dst)

```
int access_ok(int type, unsigned long addr, unsigned
   long size)
```

Returns nonzero if the userspace block of memory is valid and zero otherwise

Large-size Allocations

- Typically used when adding large-size data structures to the kernel in a stable way
- This is the case when, e.g., mounting external modules
- The main APIs are:
  - `void *vmalloc(unsigned long size)`

    allocates memory of a given size, which can be non-contiguous, and returns the virtual address (the corresponding frames are reserved)
  - `void vfree(void *addr)`

    frees the above mentioned memory

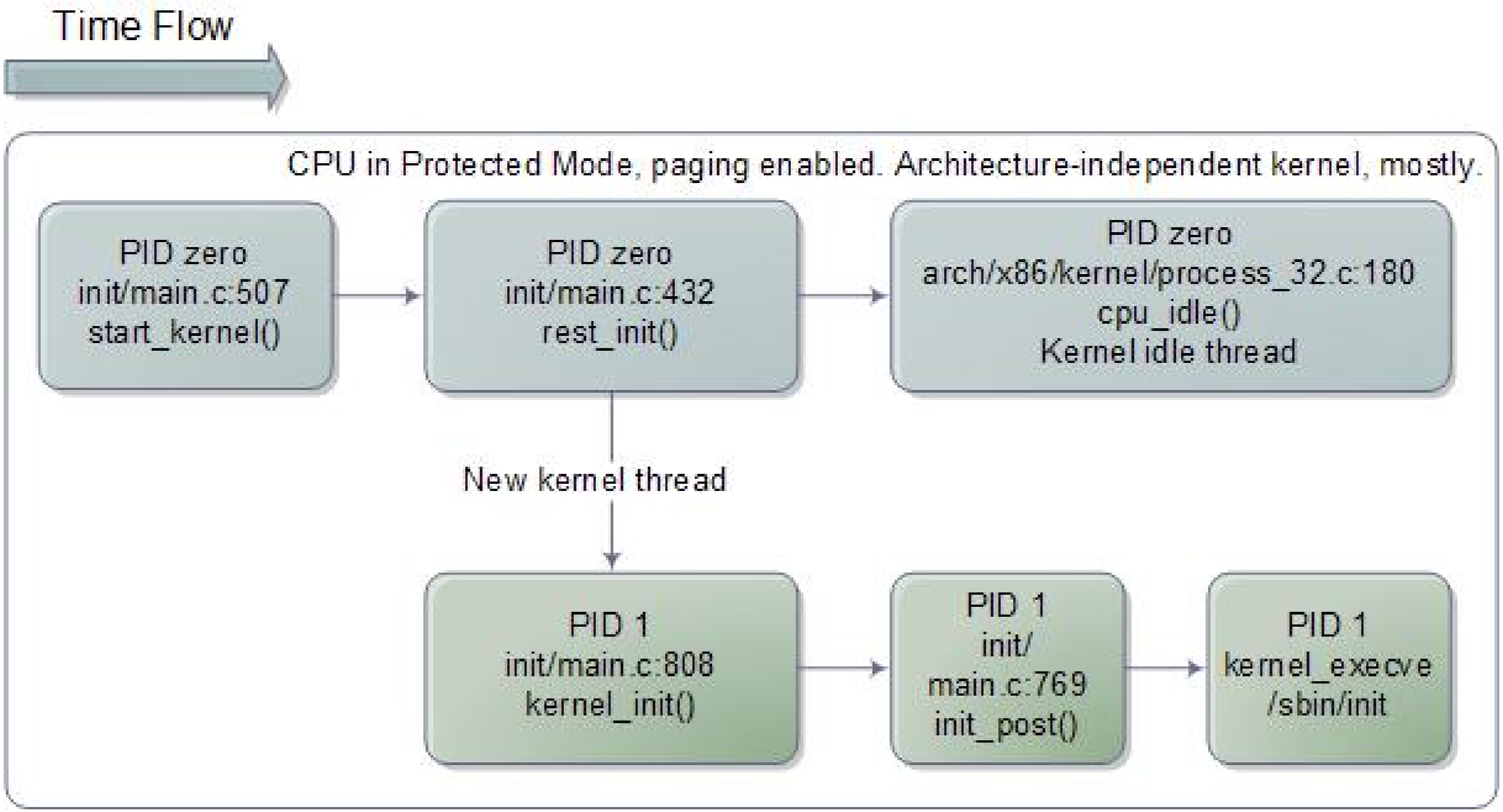
Logical/Physical Address Translation

- This is valid only for kernel directly mapped memory (not vmalloc'd memory)
- `virt_to_phys(unsigned int addr)` (in `include/x86/io.h`)
- `phys_to_virt(unsigned int addr)` (in `include/x86/io.h`)

`kmalloc()` **VS** `vmalloc()`

- Allocation size:
  - Bounded for kmalloc (cache aligned)
    - The boundary depends on the architecture and the Linux version. Current implementations handle up to 8KB
  - 64/128 MB for vmalloc
- Physical contiguousness
  - Yes for kmalloc
  - No for vmalloc
- Effects on TLB
  - None for kmalloc
  - Global for vmalloc (transparent to vmalloc users)

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`

trap_init()

```
gate_desc idt_table[NR_VECTORS] __page_aligned_bss;

void __init trap_init(void) {
        set_intr_gate(X86_TRAP_DE, divide_error);
        set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
        set_system_intr_gate(X86_TRAP_OF, &overflow);
        set_intr_gate(X86_TRAP_BR, bounds);
        set_intr_gate(X86_TRAP_UD, invalid_op);
        set_intr_gate(X86_TRAP_NM, device_not_available);
        set_task_gate(X86_TRAP_DF, GDT_ENTRY_DOUBLEFAULT_TSS);
        set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
        set_intr_gate(X86_TRAP_OLD_MF, coprocessor_segment_overrun);
        set_intr_gate(X86_TRAP_TS, invalid_TSS);
        ...
        set_system_trap_gate(SYSCALL_VECTOR, &system_call);
        ...
}
```

0x80

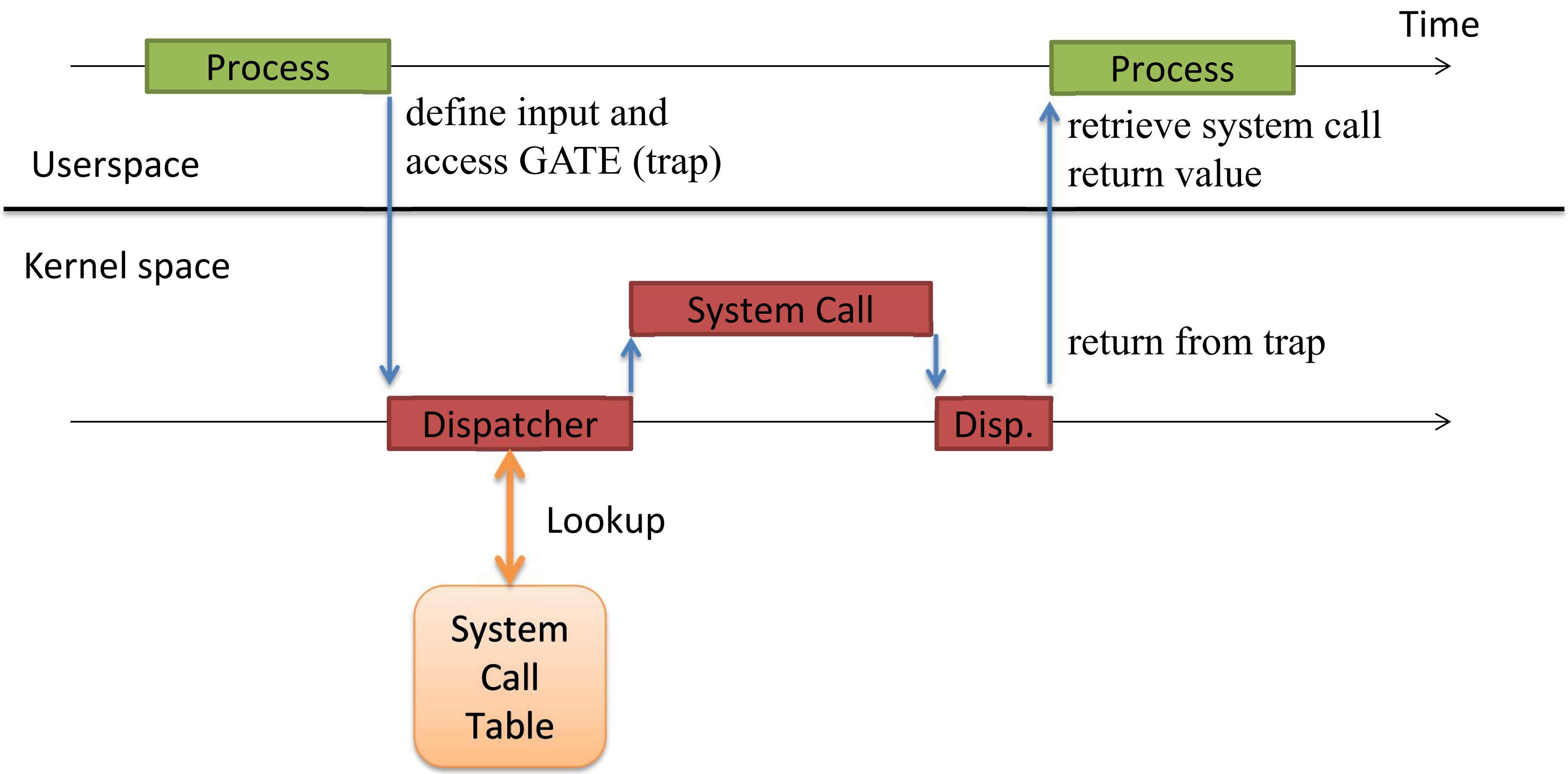
Userspace Kernel API: System Calls

- For Linux (same for Windows), the gate for on-demand access (via software traps) to the kernel is only one
- For i386 machines the corresponding software traps are:
  - 0x80 for LINUX
  - 0x2E for Windows
- The software module associated with the on-demand access GATE implements a *dispatcher* that is able to trigger the activation of the specific system call targeted by the application

System Call Dispatching

- The main data structure is the *system calls* table
- Each entry of the table points to a kernel-level function, activated by the dispatcher
- To access the correct entry, the dispatcher needs as input the system call number (provided in a CPU register)
- The code is used to identify the target entry within the system call table
- The system call is activated via an indirect `call`
- The return value is returned in a register

Dispatcher Mechanism



Traps vs. Interrupts

- Differently from interrupts, trap management does not automatically reset the interruptible-state of a CPU core (IF)
- Critical sections in the trap handler must explicitly mask and then re-enable interrupts (`cli` and `sti` innstructions)
- For SMP/multi-core machines this **might not be enough** to guarantee correctness (atomicity) while handling the trap
- The kernel uses spinlocks, based on atomic test-end-set primitives
  - We have already seen an example of CAS based on `cmpxchg`
  - Another option is the `xchg` instruction

Predefined Syscall Interface (2.4)

- This is all based on macros
  - Macros for standard formats are in `include/asm-xx/unistd.h` (or `asm/unistd.h`)
- There we find:
  - System call numerical codes
    - They are numbers used to invoke a syscall for userspace
    - They are a displacement in the syscall table for kernel space
  - Standard macros to let userspace access the gate to the Kernel
    - There is a macro for each range of parameters, from 0 to 6

Syscall codes (2.4.20)

```
/*
 * This file contains the system call numbers.
 */

#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
#define __NR_write          4
#define __NR_open               5
#define __NR_close          6
#define __NR_waitpid        7
#define __NR_creat          8
#define __NR_link               9
#define __NR_unlink         10
#define __NR_execve         11
#define __NR_chdir          12
………
#define __NR_fallocate      324
```

Macro for a 0-Parameters Syscall

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name)); \
__syscall_return(type,__res); \
}
```

Example syscall: `fork()`

Return from a syscall

```
/* user-visible error numbers are in the range -1 - -124:
  see <asm-i386/errno.h> */

#define __syscall_return(type, res) \
do { \
       if ((unsigned long)(res) >= (unsigned long)(-125)) { \
              errno = -(res); \
              res = -1; \
       } \
       return (type) (res); \
} while (0)
```

Only if `res` in [–1, -124]

What's that?!

Macro for a 1-Parameter Syscall

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
      : "=a" (__res) \
      : "0" (__NR_##name),"b" ((long)(arg1))); \
__syscall_return(type,__res); \
}
```

Example syscall: `close()`

Macro for a 6-Parameters Syscall

```
#define _syscall6(type,name,type1,arg1,type2,arg2,\
                  type3,arg3,type4,arg4,type5,arg5,type6,arg6) \
type name (type1 arg1,type2 arg2,type3 arg3,\
           type4 arg4,type5 arg5,type6 arg6) \
{ \
      long __res; \
      __asm__ volatile (
            "push %%ebp ; movl %%eax,%%ebp ;"\
            "movl %1,%%eax ; int $0x80 ; pop %%ebp" \
            : "=a" (__res) \
            : "i" (__NR_##name),"b" ((long)(arg1)),\
              "c" ((long)(arg2)),"d" ((long)(arg3)),\
              "S" ((long)(arg4)),"D" ((long)(arg5)), \
              "0" ((long)(arg6))
      ); \
      __syscall_return(type,__res); \
}
```

i386 Calling Conventions (syscalls)

```
/*
 *       0(%esp) - %ebx        ARGS
 *       4(%esp) - %ecx
 *       8(%esp) - %edx
 *       C(%esp) - %esi
 *      10(%esp) - %edi
 *      14(%esp) - %ebp        END ARGS
 *      18(%esp) - %eax
 *      1C(%esp) - %ds
 *      20(%esp) - %es
 *      24(%esp) - orig_eax
 *      28(%esp) - %eip
 *      2C(%esp) - %cs
 *      30(%esp) - %eflags
 *      34(%esp) - %oldesp
 *      38(%esp) - %oldss
*/
```

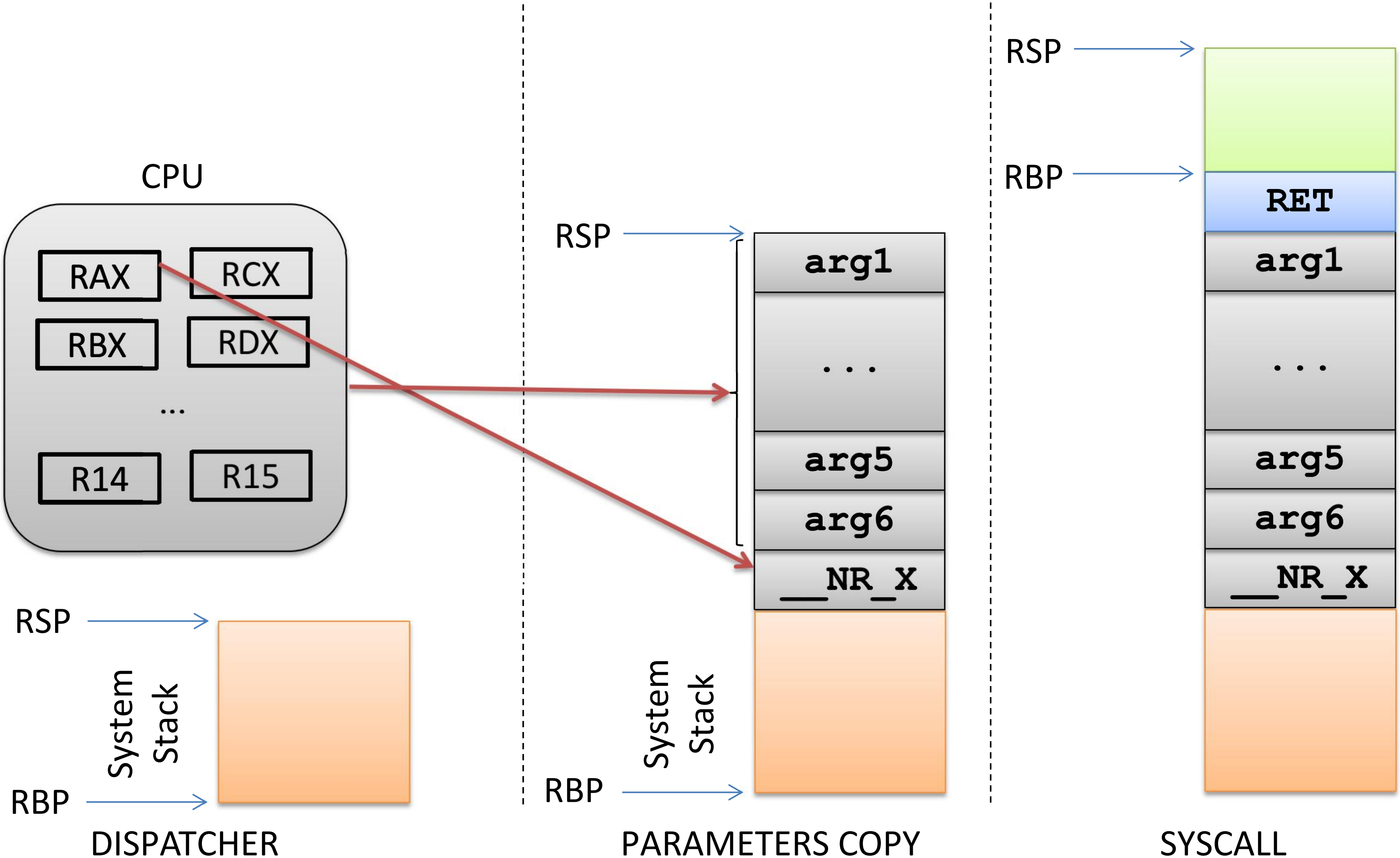
x64 Calling Conventions (syscalls)

```
/*
 * Register setup:
 * rax  system call number
 * rdi  arg0
 * rcx  return address for syscall/sysret, C arg3
 * rsi  arg1
 * rdx  arg2
 * r10  arg3 (--> moved to rcx for C)
 * r8   arg4
 * r9   arg5
 * r11  eflags for syscall/sysret, temporary for C
 * r12-r15,rbp,rbx saved by C code, not touched.
 *
 * Interrupts are off on entry.
 * Only called from user space.
 */
```

Parameters Passing

- Once gained control, the dispatcher takes a complete snapshot of CPU registers to provide the corresponding values to the actual system call
- The snapshot is taken within the system level stack
- Then the system call is invoked as a subroutine call (via a `call`)
- The system call retrieves the parameters from stack via the base pointer (remember `asmlinkage`?)

Parameters Passing



A Userspace Example

```
#include <unistd.h>
#define _NR_my_first_sys_call  254
#define _NR_my_second_sys_call  255

_syscall0(int,my_first_sys_call);
_syscall1(int,my_second_sys_call,int,arg);
```

Limitations

- The syscall has a maximum number of entries
- Resizing it requires reshuffling the whole kernel compilation process … why?
- There are few entries free. Example with Kernel 2.4.25:
  - The maximum number of entries is specified by the macro: `#define _NR_syscalls 270` in `include/linux/sys.h`
  - As specified by `include/asm-i386/unistd.h`, the available system call numerical codes start at 253
  - By default, we have space from 253 to 269

`syscall()`

- This is a construct introduced in Kernel 2.6 for the Pentium 3 chip
- Implemented in glibc (stdlib.h)
- It triggers a trap to to execute a generic system call
- The first argument is the system call number
- The other parameters are the input for the system call code
- Based on new x86 instructions: `sysenter`/`sysexit` or `syscall`/`sysret` (initially for AMD chips)

An example

```
#include <stdlib.h>

#define __NR_my_first_sys_call  333
#define __NR_my_second_sys_call 334

int my_first_sys_call(){
	return syscall(__NR_my_first_sys_call);
}

int my_second_sys_call(int arg1){
	return syscall(__NR_my_second_sys_call, arg1);
}

int main(){
	int x;

	my_first_sys_call();
	my_second_sys_call(x);
}
```

Using `syscall` or `int $0x80` explicitly

EXAMPLE SESSION

The syscall Table

- The kernel level system call table is defined in specific files
  - For Kernel 2.4.20 on i386 it is defined in `arch/i386/kernel/entry.S`
  - For kernel 2.6 is in `arch/x86/kernel/syscall_table32.S`
- These files contain preprocessor ASM directives
- Entries keep a reference to the kernel-level system call implementation
- Typically, the kernel-level name resembles the one used at application level
- In some version of the tree, the gate is also there (LXR is your friend here!)

The syscall Table

```
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall) /* 0  -  old "setup()" 
system call*/
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
        .long SYMBOL_NAME(sys_open)                 /* 5 */
        .long SYMBOL_NAME(sys_close)
        ……
        .long SYMBOL_NAME(sys_sendfile64)
        .long SYMBOL_NAME(sys_ni_syscall) /* 240 reserved */
        ………
        .long SYMBOL_NAME(sys_ni_syscall) /* 252 */
```

Place new syscalls here!

```
        .rept NR_syscalls-(.-sys_call_table)/4
                .long SYMBOL_NAME(sys_ni_syscall)
        .endr
```

Defining a new syscall

- For the previous example, the syscall entry should be:

```
.long SYMBOL_NAME(sys_my_first_sys_call)
.long SYMBOL_NAME(sys_my_second_sys_call)
```

- The code of new system calls (generally only C code) is included in any C file in the tree (possibly a new one)
- The code can use any kernel data structure and any kernel-level function (of course, except for `static` functions)
- Remember `asklinkage`!

Syscall Dispatcher (i386)

```
ENTRY(system_call)
      pushl %eax  # save orig_eax
      SAVE_ALL
      GET_CURRENT(%ebx)
      testb $0x02,tsk_ptrace(%ebx)  # PT_TRACESYS
      jne tracesys
      cmpl $(NR_syscalls),%eax
      jae badsys
      call *SYMBOL_NAME(sys_call_table)(,%eax,4)
      movl %eax,EAX(%esp)     # save the return value
ENTRY(ret_from_sys_call)
      cli         # need_resched and signals atomic test
      cmpl $0,need_resched(%ebx)
      jne reschedule
      cmpl $0,sigpending(%ebx)
      jne signal_return
restore_all:
      RESTORE_ALL
```

Fast syscall Path

SYSENTER for 32 bits - SYSCALL for 64 bits

based on model-specific registers

CS register set to the value of (SYSENTER_CS_MSR)

EIP register set to the value of (SYSENTER_EIP_MSR)

SS register set to the sum of (8 plus the value in SYSENTER_CS_MSR)

ESP register set to the value of (SYSENTER_ESP_MSR)

SYSEXIT for 32 bits - SYSRET for 64 bits

based on model-specific registers

CS register set to the sum of (16 plus the value in SYSENTER_CS_MSR)

EIP register set to the value contained in the EDX register

SS register set to the sum of (24 plus the value in SYSENTER_CS_MSR)

ESP register set to the value contained in the ECX register

Model-Specific Registers for syscalls

/usr/src/linux/include/asm/msr.h:

```
#define MSR_IA32_SYSENTER_CS   0x174
#define MSR_IA32_SYSENTER_ESP 0x175
#define MSR_IA32_SYSENTER_EIP  0x176
```

/usr/src/linux/arch/x86/kernel/sysenter.c:

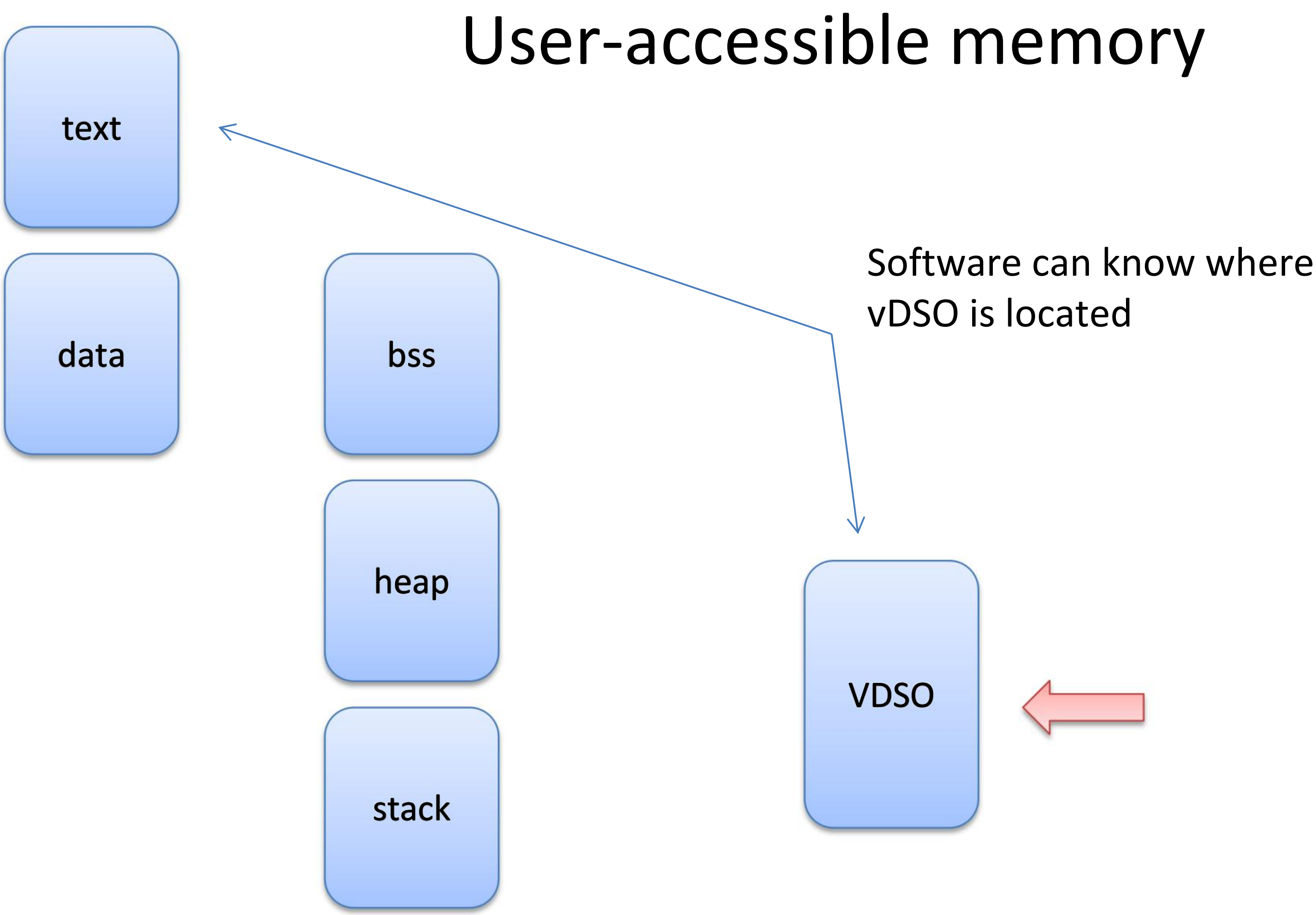
```
wrmsr(MSR_IA32_SYSENTER_CS, __KERNEL_CS, 0);
wrmsr(MSR_IA32_SYSENTER_ESP, tss->esp1, 0);
wrmsr(MSR_IA32_SYSENTER_EIP, (unsigned long) sysenter_entry, 0);
```

Again based on `rdmsr` and `wrmsr`

Virtual Dynamic Shared Object (vDSO)

- Syscall entry/exit points are set by the Kernel
- Few memory pages are created and made visible to all processes' addres spaces when they are initialized
- There processes find the actual code for the syscall entry/exit mechanism
- For i386 the definition is (up to Kernel 2.6.23) in `arch/i386/kernel/vsyscall-sysenter.S`
- In later versions, it's become an actual shared library. The source tree is at `/source/arch/x86/vdso` and the entry point is thus moved to `/arch/x86/vdso/vdso32/sysenter.S`

Mapping vDSO



Exposing vDSO

```
#include <sys/auxv.h>
void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR);
```

The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the vDSO.

Extracting vDSO from a running process

EXAMPLE SESSION

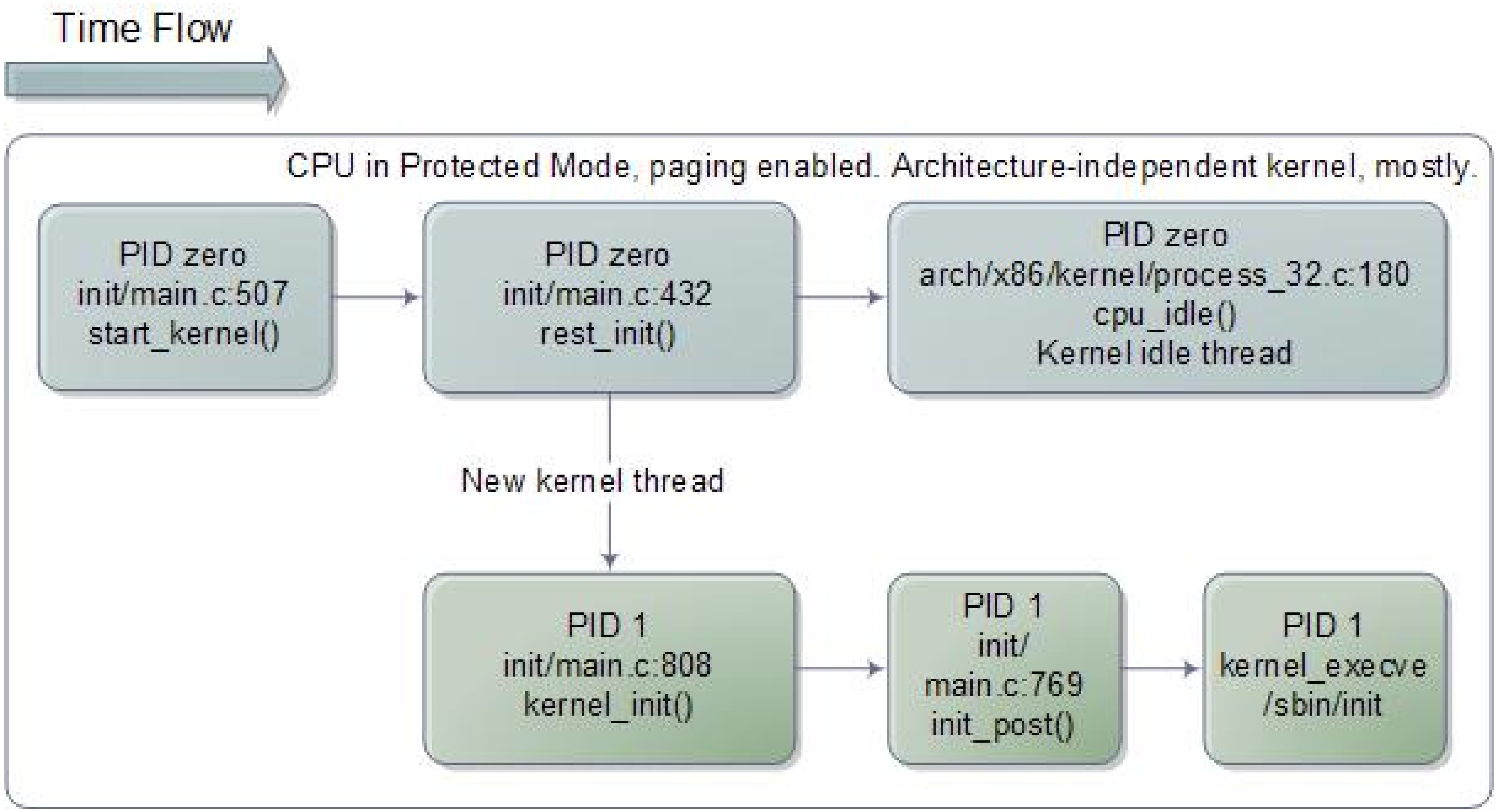
vDSO Entry Point

```
__kernel_vsyscall:
        push %ecx
        push %edx
        push %ebp
        movl %esp,%ebp
        sysenter
        nop
        /* 14: System call restart point is here! */
        int $0x80
        /* 16: System call normal return point is here! */
        pop %ebp
        pop %edx
        pop %ecx
        ret
```

Performance Aspects

- The vDSO Kernel entry point exploits flat addressing to bypass segmentation and the related operations
- It therefore reduces the number of accessed to memory in order to support the change to kernel mode
- Studies show that the reduction of clock cycles for system calls can be on the order of 75%
- It Allows randomization: security is enhanced

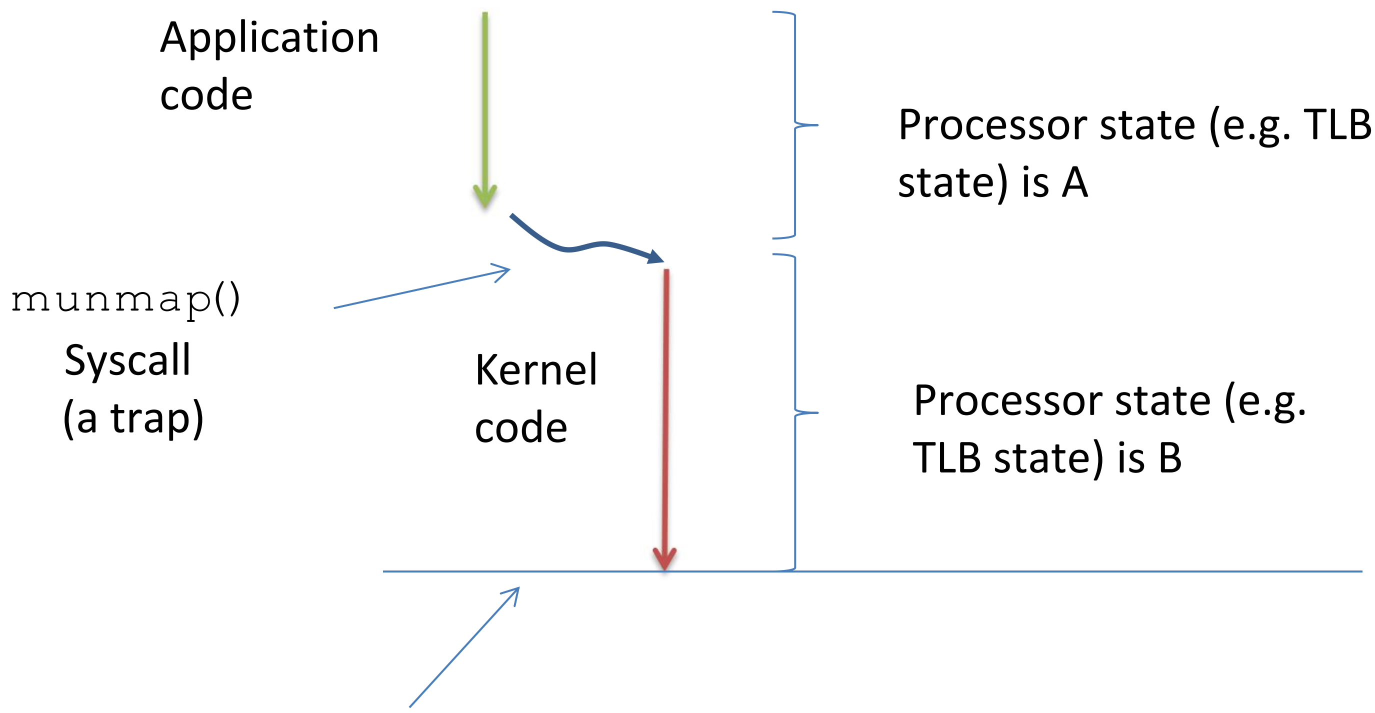
Back to Interrupt Management



Interrupts on Single-Core Machines

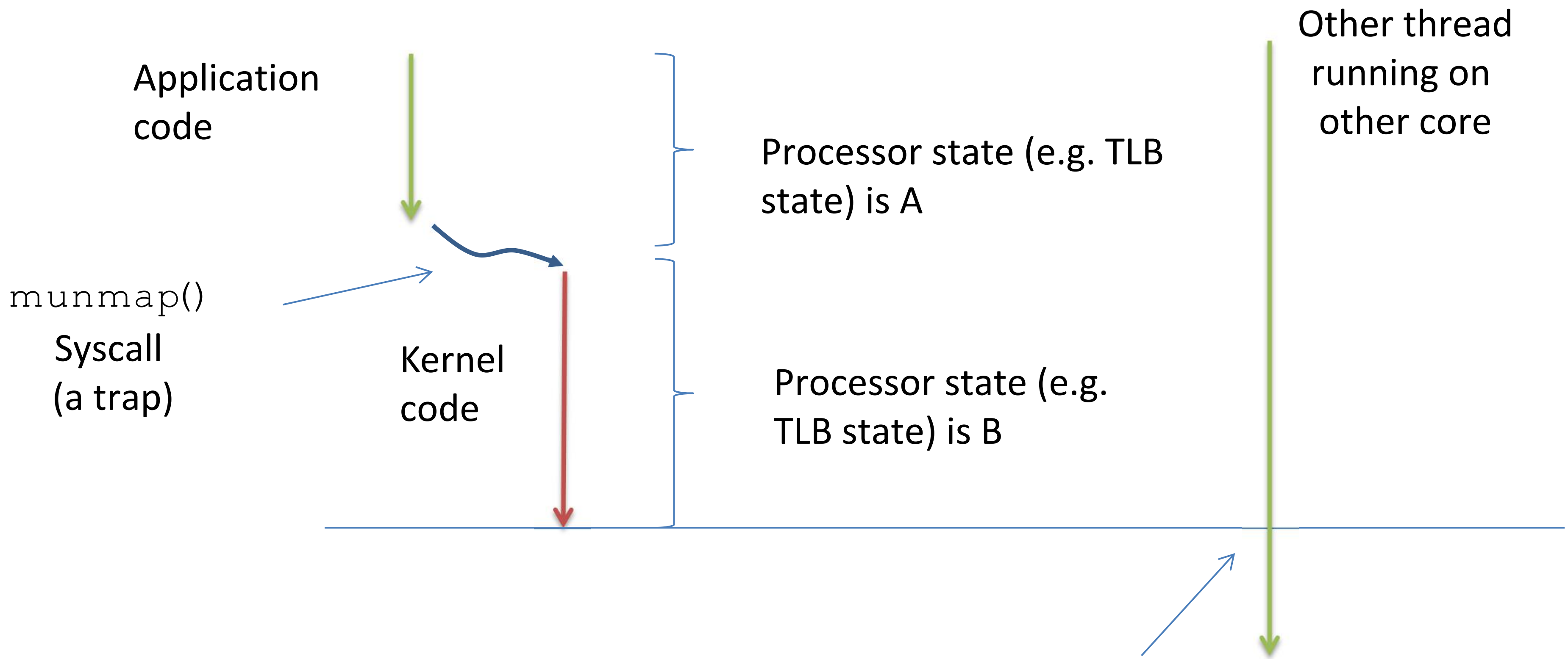
- Traditional single-core machines only relied on:
  - Traps (synchronous events wrt software execution)
  - Interrupts from external devices (asynchronous events)
- The classical way of handling the event was based on running operating system code on the single core in the system
- This was enough (in terms of consistency) even for individual concurrent (multi-thread) applications given that the state of the hardware was time-shared across threads

An Example



from this time instant, any time-shared thread sees the correct final state as determined by trap handling

An Example on Multi-cores



This thread does not see state B – what if the TLB on core 1 caches the same page table as core 0?

Main Issues

- If the system state is replicated in the architecture, we need to propagate changes by traps/interrupts
- A trap on core 0 must be propagated to core 1
- In some cases this is addressed by firmware protocols (such as when the event is bound to deterministic handling)
- Otherwise we need mechanisms to propagate and handle the event at the operating system (software) level

Inter Processor Interrupts

- IPI is a third type of event (beyond traps and classical interrupts) that may trigger the execution of specific operating-system software on any core
- An IPI is synchronous at the sender core and asynchronous at the receiver core
- IPI is typically used to enforce cross-core activities (e.g. request/reply protocols) allowing a specific core to trigger a change in the state of another

IPIs

- IPIs are generated at firmware level, but are processed at software level
- At least two priority levels are available: High and Low
- High priority leads to immediate processing of the IPI at the recipient (a single IPI is accepted and stands out at any point in time)
- Low priority generally lead to queueing the requests and process them in a serialized way

Hardware Support on x86

- We have already seen the registers to trigger IPIs
- They are an interface to the APIC/LAPIC circuitry
- LAPIC offers an instance local to any core
- LAPIC is where the programmable timer is (for time tracking and time-sharing purposes)
- IPI requests travel along an ad-hoc APIC bus
  - On modern x86 architectures, this is the QuickPath Interconnect
  - Again, go to the DCHPC courses for more info on this

IDT Entries

Vector range	**Use**
0-19 (0x0-0x13)	Nonmaskable interrupts and exceptions
20-31 (0x14-0x1f)	Intel-reserved
32-127 (0x20-0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls (segmented style)
129-238 (0x81-0xee)	External interrupts (IRQs)
239 (0xef)	**Local APIC timer interrupt**
240-250 (0xf0-0xfa)	Reserved by Linux for future use
251-255 (0xfb-0xff)	**Interprocessor interrupts**

More on IDT Initialization

- We already mentioned `trap_init()`
- `init_IRQ()` in `arch/x86/kernel/irqinit.c` takes care of setting up device interrupts (the latter is based on ACPI)
- Main functions to setup the IDT:
  - `set_trap_gate()` initializes one IDT entry to define the value 0 as the privilege level admitted for accessing the gate via software
  - `set_intr_gate()` is similar, but handler activation relies on interrupt masking
  - `set_system_gate()` is similar to `set_trap_gate()` but it defines the value 3 as the privilege level to access the gate

Initialization on x64

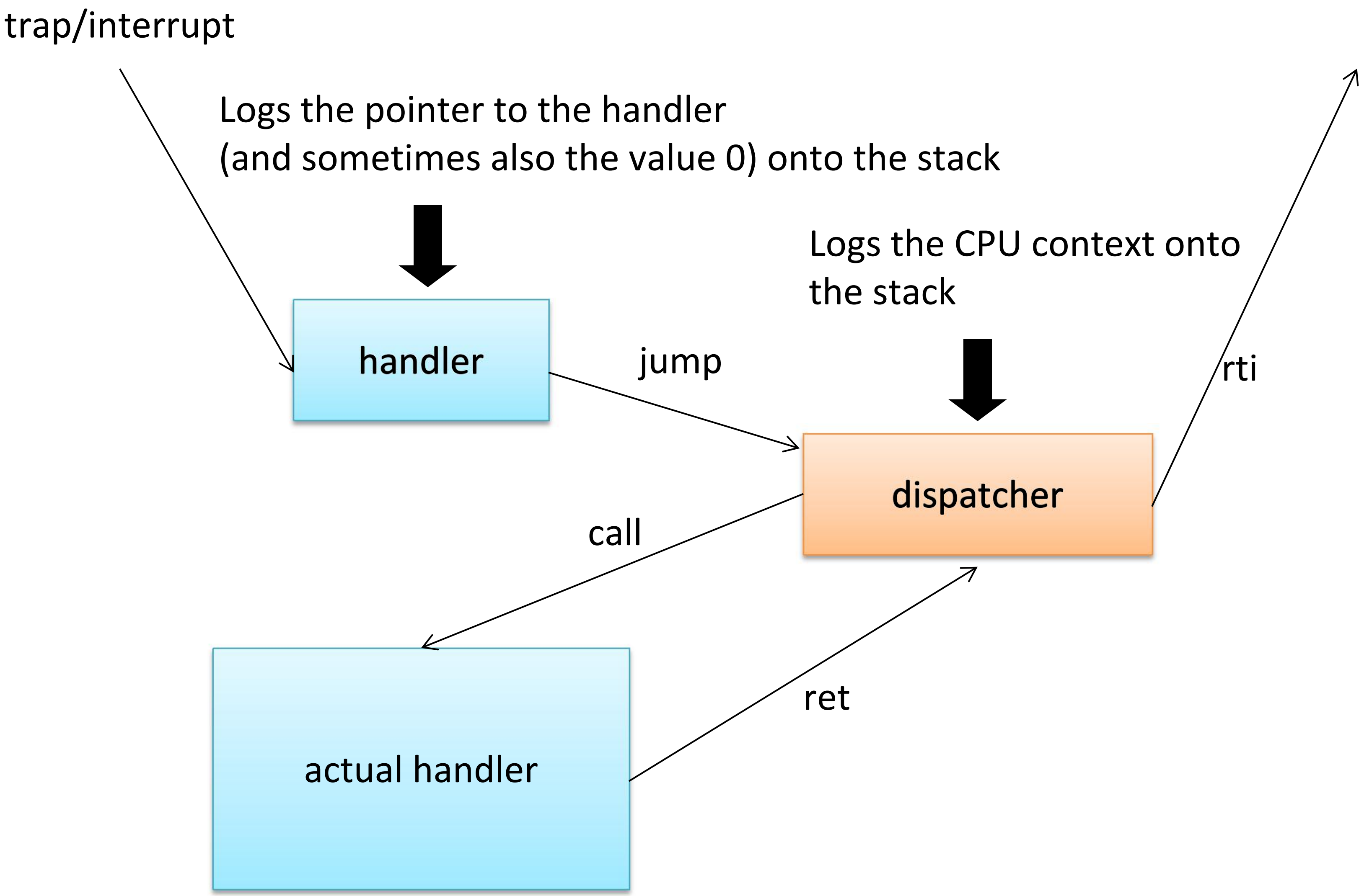
CODE SNIPPET FROM desc.h

```
/*
 * This routine sets up an interrupt gate at directory privilege level 3.
 */
static inline void set_system_intr_gate(unsigned int n, void *addr)
{
	BUG_ON((unsigned)n > 0xFF);
	_set_gate(n, GATE_INTERRUPT, addr, 0x3, 0, __KERNEL_CS);
}

static inline void set_system_trap_gate(unsigned int n, void *addr)
{
	BUG_ON((unsigned)n > 0xFF);
	_set_gate(n, GATE_TRAP, addr, 0x3, 0, __KERNEL_CS);
}

static inline void set_trap_gate(unsigned int n, void *addr)
{
	BUG_ON((unsigned)n > 0xFF);
	_set_gate(n, GATE_TRAP, addr, 0, 0, __KERNEL_CS);
}
```

Modular handler management on i386

- Trap/interrupt handlers are defined in `arch/x86/kernel/entry_32.S` (along with the system call dispatcher)
- Handlers associated with default trap/interrupts (from 0 to 31) are managed via an additional dispatcher
- Each handler logs a null-value on the stack in case no error code is generated in relation to the specific trap/interrupt
- Then it logs on the stack the address of the actual handler (typically written in C)
- After, the assembly dispatcher is activated: it logs the CPU context and gives control to the handler via a conventional call
- Input parameters are passed via the stack: `asmlinkage`!

Activation Scheme



Examples

```
ENTRY(overflow)
      pushl $0
      pushl $ SYMBOL_NAME(do_overflow)
      jmp error_code

ENTRY(general_protection)
      pushl $ SYMBOL_NAME(do_general_protection)
      jmp error_code

ENTRY(page_fault)
      pushl $ SYMBOL_NAME(do_page_fault)
      jmp error_code
```

`error_code` on i386

- error_code logs the CPU context onto the stack
- On the stack the routine populates the following data structure, defined in `include/asm-i386/ptrace.h`

```
struct pt_regs {
    long ebx;     long ecx;
    long edx; long esi;
    long edi; long ebp;
    long eax; int xds; int xes;
    long orig_eax; long eip; int xcs;
    long eflags; long esp; int xss;
}
```

- The actual handler can take as input a `pt_regs*` pointer and, if needed, an unsigned long representing the error-code

Page Fault Handler

- The page fault handler is `do_page_fault(struct pt_regs *regs, unsigned long error_code)` defined in `linux/arch/x86/mm/fault.c`
- It takes as input the error-code associated with the occurred fault
- The fault type is specified via the three least significant bits of `error_code` according to the following rules:
  - bit 0 == 0 means no page found, 1 means protection fault
  - bit 1 == 0 means read, 1 means write
  - bit 2 == 0 means kernel, 1 means user mode

Kernel Exception Handling

- When a process runs in kernel mode, it may have to access user memory passed by a untrusted process
  - `verify_area(int type, const void * addr, unsigned long size)`
  - `access_ok(int type, unsigned long addr, unsigned long size)`
- This may take an unnecessary large amount of time
- This operation takes place quite often

Kernel Exception Handling

- Linux exploits the MMU to take care of this
- If the kernel accesses an address which is not accessible, a page fault is generated
- The unaccessible address is taken from CR2
  - If the address is within the VA space of the process we either have to swap in the page or there was an access in write mode to a read-only page
- Otherwise, a jump to `bad_area` label tries to activate a *fixup*

Kernel Fixups

- In `bad_area`, the kernel uses the address in `regs->eip` to find a suitable place to recover execution
- This is done by replacing the content of `regs->eip` with the *fixup address*
- This must be executable code in kernel mode
- The fixup is defined by macros
- An example: `get_user(c, buf)` in `arch/x86/include/asm/uaccess.h` as called from `drivers/char/sysrq.c`

Fixup: Expanded Macro

```
(
  {
    long __gu_err = - 14 , __gu_val = 0;
    const __typeof__(*( (  buf ) )) *__gu_addr = ((buf));
    if (((((0 + current_set[0])->tss.segment) == 0x18 )  ||
       (((sizeof(*(buf))) <= 0xC0000000UL) &&
       ((unsigned long)(__gu_addr ) <= 0xC0000000UL - (sizeof(*(buf)))))))
      do {
        __gu_err  = 0;
        switch ((sizeof(*(buf)))) {
          case 1:
            __asm__ __volatile__(
              "1:      mov" "b" " %2,%" "b" "1\n"
              "2:\n"
              ".section .fixup,\"ax\"\n"
              "3:      movl %3,%0\n"
              "        xor" "b" " %" "b" "1,%" "b" "1\n"
              "        jmp 2b\n"
              ".section __ex_table,\"a\"\n"
              "        .align 4\n"
              "        .long 1b,3b\n"
              ".text"       : "=r"(__gu_err), "=q" (__gu_val): "m"((*(struct __large_struct *)
                           (   __gu_addr   )) ), "i"(- 14 ), "0"(  __gu_err  )) ;
              break;
          case 2:
            __asm__ __volatile__(
              "1:      mov" "w" " %2,%" "w" "1\n"
              "2:\n"
              ".section .fixup,\"ax\"\n"
              "3:      movl %3,%0\n"
              "        xor" "w" " %" "w" "1,%" "w" "1\n"
              "        jmp 2b\n"
```

Fixup: Expanded Macro

```
            ".section __ex_table,\"a\"\n"
            "        .align 4\n"
            "        .long 1b,3b\n"
            ".text"        : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct __large_struct *)
                          (   __gu_addr   )) ), "i"(- 14 ), "0"(  __gu_err  ));
            break;
        case 4:
          __asm__ __volatile__(
            "1:      mov" "l" " %2,%" "" "1\n"
            "2:\n"
            ".section .fixup,\"ax\"\n"
            "3:      movl %3,%0\n"
            "        xor" "l" " %" "" "1,%" "" "1\n"
            "        jmp 2b\n"
            ".section __ex_table,\"a\"\n"
            "        .align 4\n"        "        .long 1b,3b\n"
            ".text"        : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct __large_struct *)
                          (   __gu_addr   )) ), "i"(- 14 ), "0"(__gu_err));
            break;
        default:
          (__gu_val) = __get_user_bad();
      }
    } while (0) ;
  ((c)) = (__typeof__(*((buf))))__gu_val;
  __gu_err;
 }
);
```

Fixup: Generated Assembly

```
        xorl %edx,%edx
        movl current_set,%eax
        cmpl $24,788(%eax)
        je .L1424
        cmpl $-1073741825,64(%esp)
        ja .L1423
.L1424:
        movl %edx,%eax
        movl 64(%esp),%ebx
1:      movb (%ebx),%dl  /* this is the actual user access */
2:
.section .fixup,"ax"
3:      movl $-14,%eax
        xorb %dl,%dl
        jmp 2b
.section __ex_table,"a"
        .align 4
        .long 1b,3b
.text
.L1423:
        movzbl %dl,%esi
```

Non-standard Sections

Fixup: Linked Code

```
$ objdump --disassemble --section=.text vmlinux

 c017e785 <do_con_write+c1> xorl   %edx,%edx
 c017e787 <do_con_write+c3> movl   0xc01c7bec,%eax
 c017e78c <do_con_write+c8> cmpl   $0x18,0x314(%eax)
 c017e793 <do_con_write+cf> je     c017e79f <do_con_write+db>
 c017e795 <do_con_write+d1> cmpl   $0xbfffffff,0x40(%esp,1)
 c017e79d <do_con_write+d9> ja     c017e7a7 <do_con_write+e3>
 c017e79f <do_con_write+db> movl   %edx,%eax
 c017e7a1 <do_con_write+dd> movl   0x40(%esp,1),%ebx
 c017e7a5 <do_con_write+e1> movb   (%ebx),%dl
 c017e7a7 <do_con_write+e3> movzbl %dl,%esi
```

Fixup Sections

```
$ objdump --section-headers vmlinux

 vmlinux:     file format elf32-i386

 Sections:
 Idx Name          Size      VMA       LMA       File off  Algn
   0 .text         00098f40  c0100000  c0100000  00001000  2**4
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
   1 .fixup        000016bc  c0198f40  c0198f40  00099f40  2**0
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
   2 .rodata       0000f127  c019a5fc  c019a5fc  0009b5fc  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
   3 __ex_table    000015c0  c01a9724  c01a9724  000aa724  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
   4 .data         0000ea58  c01abcf0  c01abcf0  000abcf0  2**4
                   CONTENTS, ALLOC, LOAD, DATA
   5 .bss          00018e21  c01ba748  c01ba748  000ba748  2**2
                   ALLOC
   6 .comment      00000ec4  00000000  00000000  000ba748  2**0
                   CONTENTS, READONLY
   7 .note         00001068  00000ec4  00000ec4  000bb60c  2**0
                   CONTENTS, READONLY
```

Fixup Non Standard Sections

```
$ objdump --disassemble --section=.fixup vmlinux

 c0199ff5 <.fixup+10b5> movl   $0xfffffff2,%eax
 c0199ffa <.fixup+10ba> xorb   %dl,%dl
 c0199ffc <.fixup+10bc> jmp    c017e7a7 <do_con_write+e3>

$ objdump --full-contents --section=__ex_table vmlinux

  c01aa7c4 93c017c0 e09f19c0 97c017c0 99c017c0  ................
  c01aa7d4 f6c217c0 e99f19c0 a5e717c0 f59f19c0  ................
  c01aa7e4 080a18c0 01a019c0 0a0a18c0 04a019c0  ................
```

- Remember x86 is little endian!

```
  c01aa7c4 c017c093 c0199fe0 c017c097 c017c099  ................
  c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5  ................
  c01aa7e4 c0180a08 c019a001 c0180a0a c019a004  ................
```

Fixup Activation Steps

1. access to invalid address: c017e7a5 <do_con_write+e1> movb (%ebx),%dl
2. MMU generates exception
3. CPU calls do_page_fault
4. do page fault calls search_exception_table (regs->eip == c017e7a5);
5. search_exception_table looks up the address c017e7a5 in the exception table and returns the address of the associated fault handle code c0199ff5.
6. do_page_fault modifies its own return address to point to the fault handle code and returns.
7. execution continues in the fault handling code:
   a) EAX becomes -EFAULT (== -14)
   b) DL becomes zero (the value we "read" from user space)
   c) execution continues at local label 2 (address of the instruction immediately after the faulting user access).

Fixup in 64-bit Kernels

- First possibility: expand the table to handle 64-bit addresses
- Second possibility: represent offsets from the table itself

```
.long 1b,3b
```

```
.long (from) - .
.long (to) - .
```

```
ex_insn_addr(const struct exception_table_entry *x){
      return (unsigned long)&x->insn + x->insn;
}
```

Fixups in 4.6

- The exception table has been expanded with an additional field to keep a 32-bit address of a handler
- The handler is activated when a fixup is being activated
- In this way is possible to extend the behaviour of the Kernel-level exception handling

Back to IPIs

- Immediate handling is allowed for the case in which there are no data structures that are shared across CPU-cores that need to be accessed for the handling (stateless scenarios)
- An example is *system halt* (e.g. upon panic)
- Other usages of IPI are:
  - Execution of the same function across all the CPU-cores (exactly like the halt)
  - Change of the state of hardware components across multiple CPU-cores in the system (e.g. the TLB state)

Using IPIs: Some Examples

- `CALL_FUNCTION_VECTOR` (*vector* 0xfb)
  - Sent to all CPUs but the sender, forcing those CPUs to run a function passed by the sender. The corresponding interrupt handler is `call_function_interrupt()`. Usually this interrupt is sent to all CPUs except the CPU executing the calling function by means of the `smp_call_function()` facility function.
- `RESCHEDULE_VECTOR` (*vector* 0xfc)
  - When a CPU receives this type of interrupt, the corresponding handler, named `reschedule_interrupt()`, just acknowledges the interrupt.
- `INVALIDATE_TLB_VECTOR` (*vector* 0xfd)
  - Sent to all CPUs but the sender, forcing them to invalidate their TLBs. The corresponding handler, named `invalidate_interrupt()` flushes some TLB entries of the processor

IPIs' API

`send_IPI_all( )`

Sends an IPI to all CPUs (including the sender)

`send_IPI_allbutself( )`

Sends an IPI to all CPUs except the sender

`send_IPI_self( )`

Sends an IPI to the sender CPU

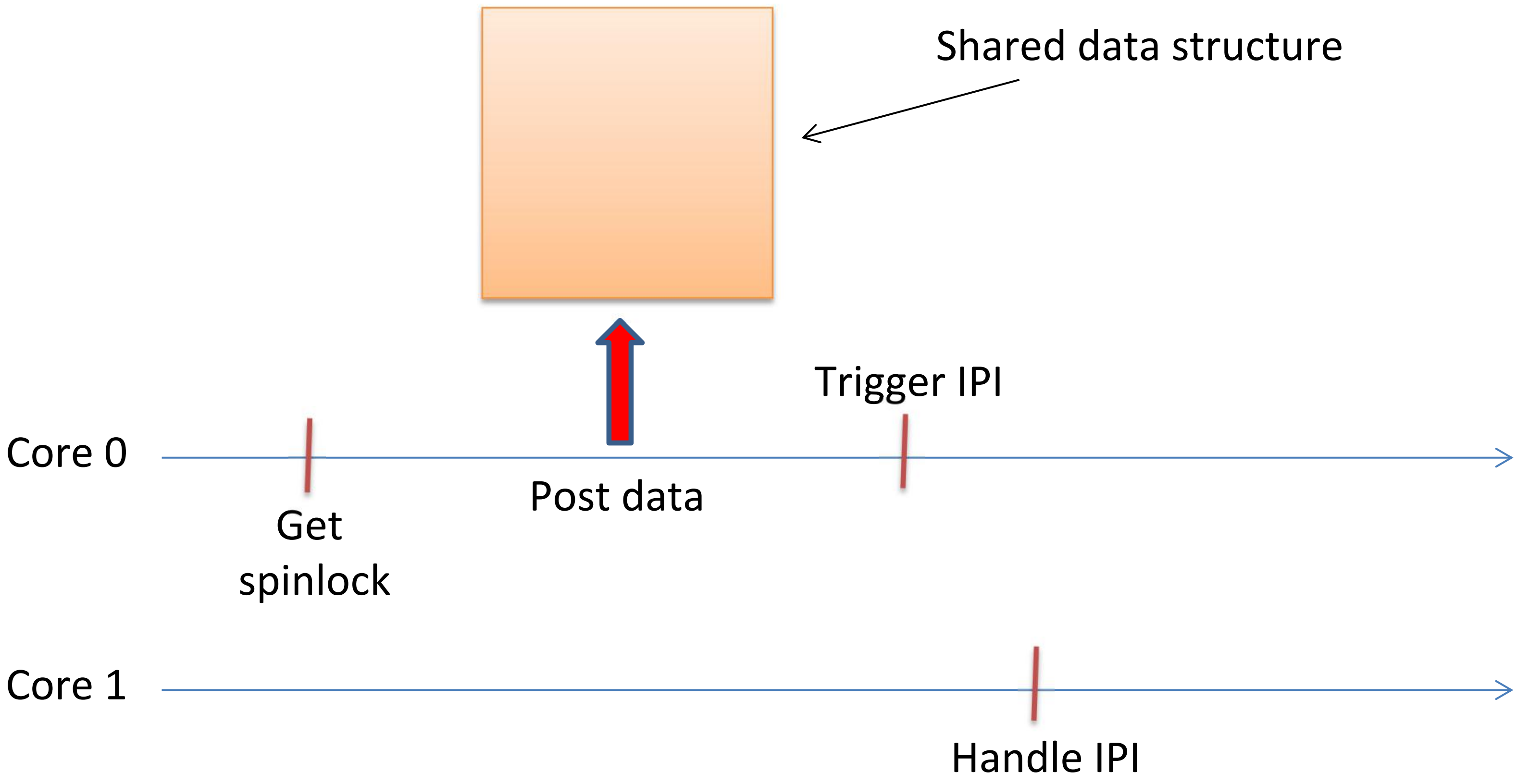
`send_IPI_mask( )`

Sends an IPI to a group of CPUs specified by a bit mask

IPI Management Sequentialization

- Sequentialization is used if the IPI needs to manage shared data structures across the threads
- For example, when an IPI requires parameters
- Parameters are actually passed into fixed memory locations (recall the Interrupt Control Registers)
- An example: SMP call function: function pointer and parameters are both passed into a global table

IPI Management Sequentialization



smp_call_function()

```
int smp_call_function(void (*_func)(void *info), void
*_info, int wait) {
        ……
        /*Can deadlock when called with interr. disabled*/
        WARN_ON(irqs_disabled());

        spin_lock_bh(&call_lock);
        atomic_set(&scf_started, 0);
        atomic_set(&scf_finished, 0);
        func = _func;
        info = _info;

        for_each_online_cpu(i)
                os_write_file(cpu_data[i].ipi_pipe[1], "C", 1);

        while (atomic_read(&scf_started) != cpus)
                barrier();
        if (wait)
                while (atomic_read(&scf_finished) != cpus)
                        barrier();

        spin_unlock_bh(&call_lock);
        return 0;
}
```

smp_call_function()

- smp_call_function()
  - arch_send_call_function_ipi_mask()
  - send_call_func_ipi()
  - native_send_call_func_ipi()
  - apic->send_IPI_mask()
  - __send_IPI_dest_field()

```
static inline void __send_IPI_dest_field(unsigned long
   mask, int vector) {
      unsigned long cfg;
      // [...]
      cfg = __prepare_ICR2(mask);
      apic_write(APIC_ICR2, cfg);
      cfg = __prepare_ICR(0, vector);
      apic_write(APIC_ICR, cfg);
}
```

An Example: Synchronize All Cores

```
static atomic_t synch_leave;
static atomic_t synch_enter;

void synchronize_all(void) {
      printk("cpu %d asking from unpreemptive
            synchronization\n", smp_processor_id());
      atomic_set(&synch_enter, num_online_cpus() - 1);
      atomic_set(&synch_leave, 1);
      preempt_disable();
      smp_call_function_many(cpu_online_mask,
            synchronize_all_slaves, NULL , false);
      while(atomic_read(&synch_enter) > 0);
      printk("cpu %d all kernel threads synchronized\n",
            smp_processor_id());
}
```

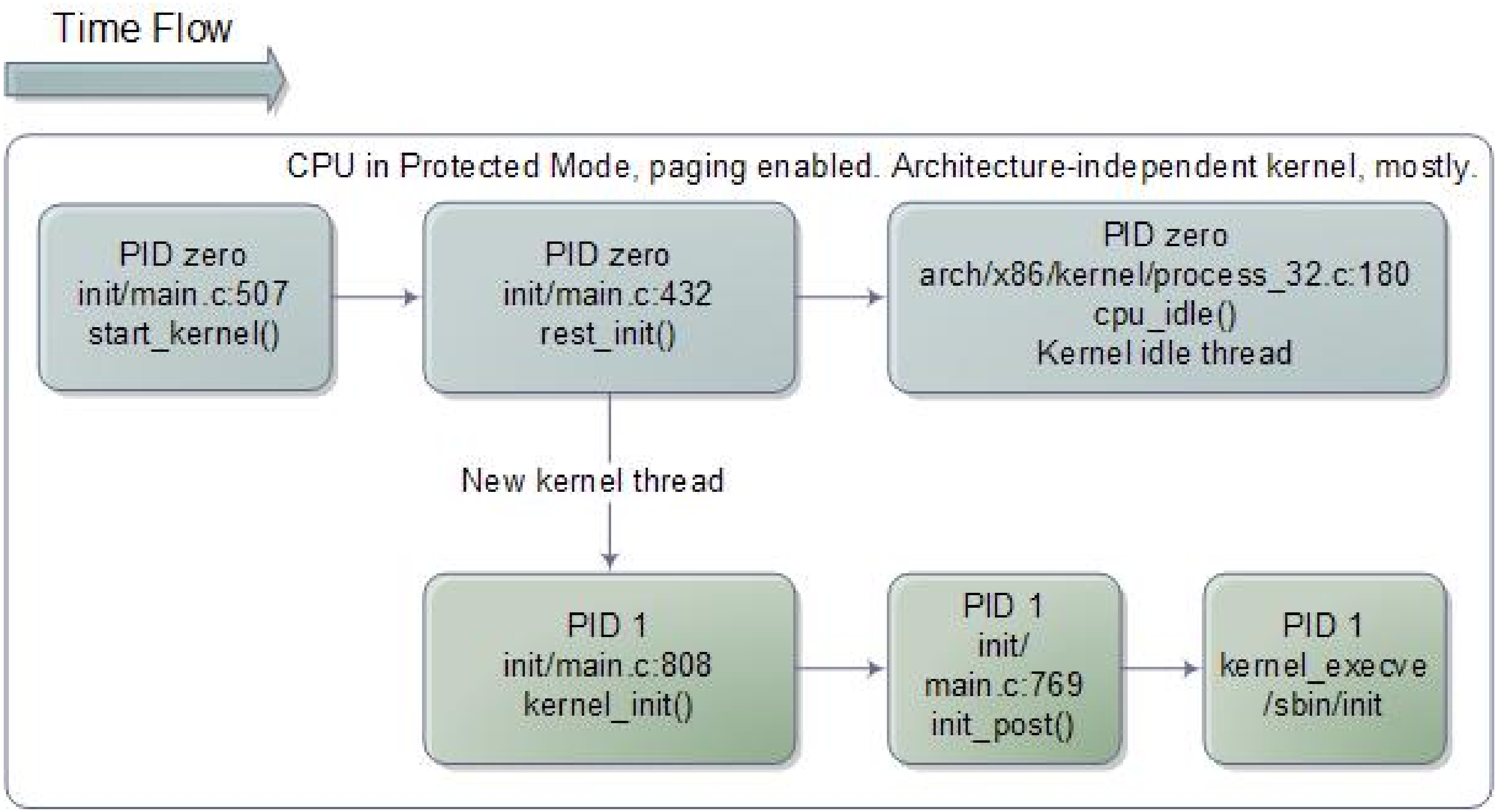
An Example: Synchronize All Cores

```
static void synchronize_all_slaves(void *info) {
       (void)info;
       printk("cpu %d entering synchronize_all_slaves\n",
                     smp_processor_id());
       atomic_dec(&synch_enter);
       preempt_disable();
       while(atomic_read(&synch_leave) > 0);
       preempt_enable();
       printk("cpu %d leaving synchronize_all_slaves\n",
              smp_processor_id());
}

void unsynchronize_all(void) {
       printk("cpu %d freeing other kernel threads\n",
                     smp_processor_id());
       atomic_set(&synch_leave, 0);
       preempt_enable();
}
```

Be careful

- IPI is an extremely powerful technology
- However you need to consider scalability aspects
- IPI-based synchronization involving large counts of cores must be used only when mandatorily needed
- The classical example is when patching the kernel on line, e.g. upon mounting a module

Back to Kernel Initialization



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 a setup system up and running, which is spinning forever
- This is the end of the "romantic" Kernel boot procedure: we infinitely loop into a `hlt` instruction
- or...