Advanced Operating Systems (and System Security)
MS degree in Computer Engineering
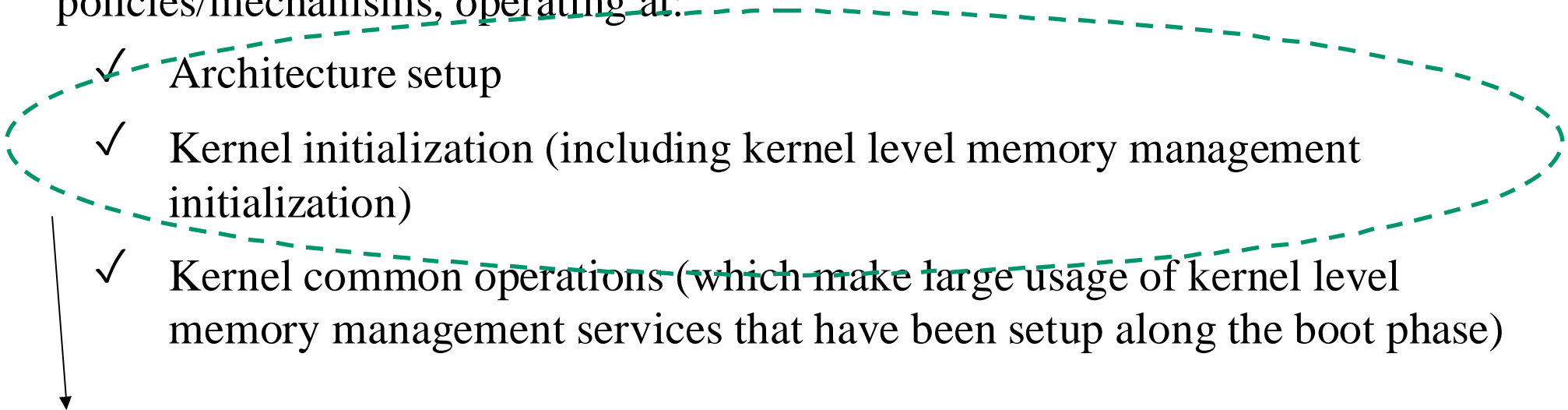## University of Rome Tor Vergata
Lecturer: Francesco Quaglia

Kernel level memory management
1. The very base on boot vs memory management
2. Memory 'Nodes' (UMA vs NUMA)
3. x86 paging support
4. Memory allocators
5. Boot and steady state behavior of the memory management system in the Linux kernel

Memory management vs system startup

- System-startup steps lead to change the image of data/code that we have in memory

- These changes need to happen just based on various memory handling policies/mechanisms, operating at:
  - ✓ Architecture setup
  - ✓ Kernel initialization (including kernel level memory management initialization)
  - ✓ Kernel common operations (which make large usage of kernel level memory management services that have been setup along the boot phase)

An initially loaded kernel memory-image is not the same as the one that common applications ``see'' when they are activated

Basic aspects to consider

- Where do we load the actual kernel image in physical memory?

  – Randomization?

- How do we make this "physical memory image" reachable using virtual addresses?

  – Randomization?

- How do we deal with kernel level information (e.g. routines) that we only need at kernel startup?

  – They just become useless at some point in time

- What are the actual kernel level mechanisms for managing memory at steady state (after boot)?

Loading the kernel to physical memory

- The target physical memory region is determined by the boot-loader

- Randomization can be put in place, meaning that a given initial zone of RAM memory is left as an unused hole

  - How do we make the actual kernel reach its information?

  - Recall that kernel software typically executes using virtual addresses

RAM address 0x0
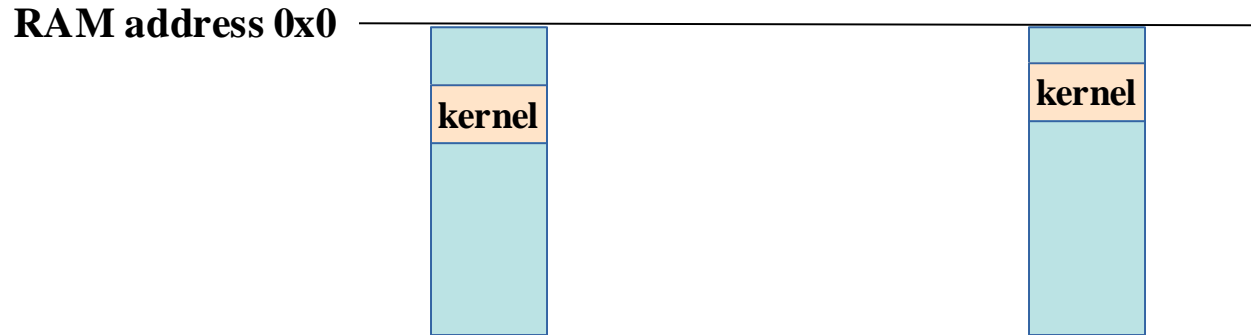
kernel

kernel

At startup we need to use/organize a correct page table to reach the target <u>physical</u> memory zone

Making the kernel reach itself via logical addresses

- The target logical memory region is determined by the boot-loader

- Randomization can be put in place, meaning that a given initial zone of logical memory is left as an unused hole

  - How do we make the actual kernel reach its information?

Reference kernel start virtual address 0xff...

kernel

kernel

At startup we need to use/organize a correct page table to reach the target <u>virtual</u> memory zone

One (typical) way to proceed

- The kernel image loaded in memory at physical address P-ADDR has a page table in its image

- This pagetable is somehow updated in order to map to P-ADDR the virtual address V-ADDR selected for the kernel startup image

- After this is done, the kernel can start using that page table for its operations

- NOTE: **we need to know the position (the offset)** of the kernel page table into the kernel image, this imposes that kernel compiling has some limits, e.g. in terms of possibility to expand data structures

Kernel page table

Physical address = P_ADDR + offset

V-ADDR → **Map to P-ADDR**
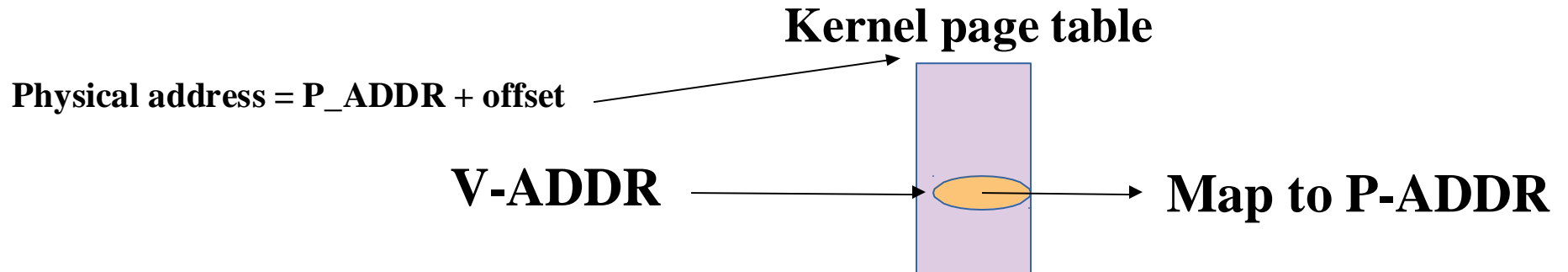
Page table identities

- The kernel page table to be generated/finalized at kernel startup is typically know as **identity page table**

- The boot loader code exploits another page table to reach memory and the booting kernel image, which is called **trampoline page table**

Trampoline page table

Address 0x0

Kernel page table

RAM
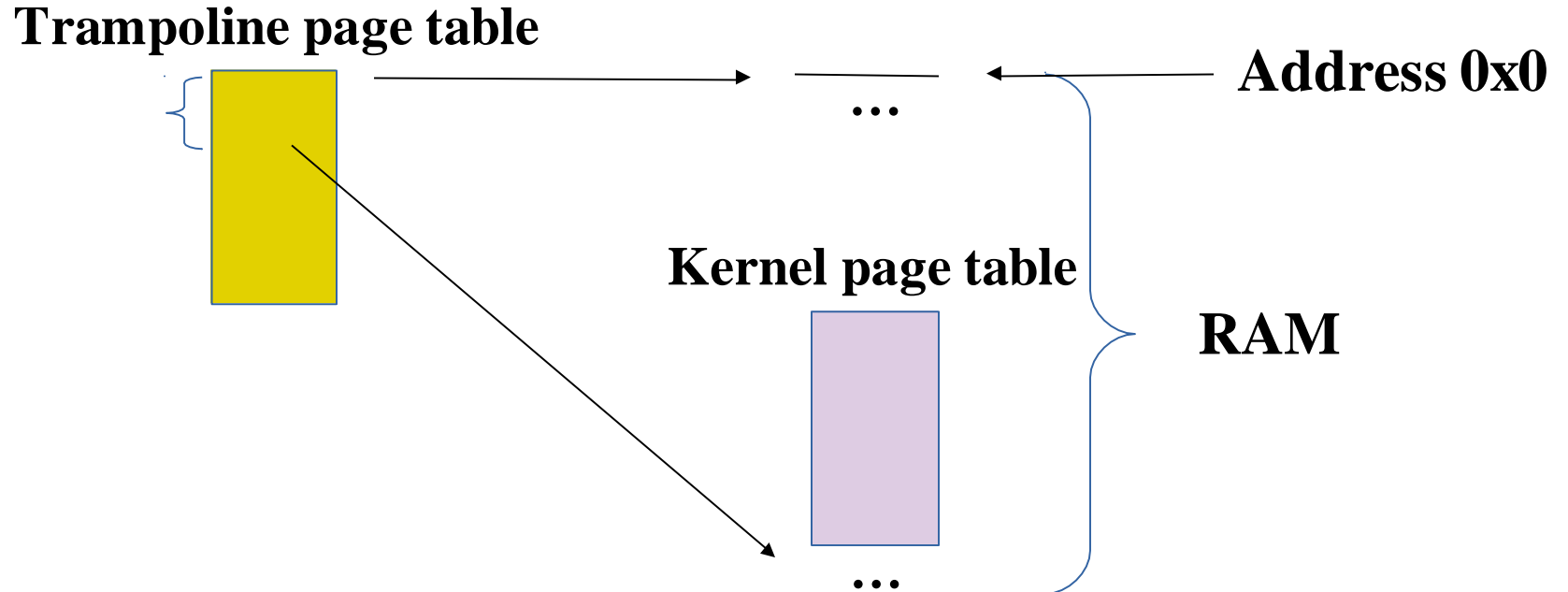
Details on the boot loader

- When the system boots, the **bootloader** (such as GRUB, LILO, or UEFI) loads the compressed kernel image (e.g., **vmlinuz**) into memory

- A small portion of the kernel, called the **decompressor** or **bootstub**, is located at the start of the kernel image and is responsible for decompressing the rest of the kernel

- The bootloader transfers control to this decompressor, which decompresses the main kernel into memory

- The trampoline page table is exactly used by the decompressor in order to reach physical and logical memory for setting up the kernel image and its identity page table

Why do we actually need physical and logical address randomization

- Randomization of the logical address of some kernel object allows protecting against kernel level attackers that attempt to use that object

- The reference to the object is based on virtual addresses

- However, if the attacker knows that the physical address for that object is fixed (not randomized) then he/she can change the identity page table for reaching that physical zone with a specific virtual address he/she selects

- Randomizaton of the physical address can protect against this attack scenario

Randomization support in Linux

- The configuration parameter CONFIG_RANDOMIZE_BASE enables compiling the kernel with both virtual and physical memory randomization

- Originally supported in Linux kernel 3.10 (year 2013)

- Randomization places:

  - Physical kernel positioning wherever in RAM memory starting from 16MB

  - Logical kernel positioning between 16MB and 1 GB from the compile time base

- Randomization can be excluded at startup time via the nokaslr option

A baseline example case – non-randomized addresses

- The <u>physical address</u> of the kernel page table (the identity one) is known at compile time

- The <u>logical address</u> of the kernel page table is known at compile time (as well as any other kernel portion)

- The startup code for paging (virtual to physical memory translation) can simply <u>set the page table pointer register of the CPU with the compile time known physical counterpart of that virtual address</u>

  - <u>That page table already redirects to the compile time known physical memory that hosts the kernel image</u>

- Then paging can be started up on the processor

- With this solution the startup code for paging is extremely simple and can even work without virtual addresses to load the kernel in memory and to setup the state of the CPU right before kernel specific startup

An example head.S code snippet - triggering Linux paging (IA32 case)

```
/* * Enable paging */ 3:
movl $swapper_pg_dir-__PAGE_OFFSET,%eax
movl %eax,%cr3 /* set the page table pointer.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0 /* ..and set paging (PG) bit */
```

An example head_64.S code snippet - triggering Linux paging (x86-64 case /kernel5)

```
/* Form the CR3 value being sure to include the CR3 modifier */
addq  $(early_top_pgt  -  __START_KERNEL_map),  %rax

……….. /* we are accounting for other stuff  */

movq %rax, %cr3
```

Compile time symbols for accessing the page table
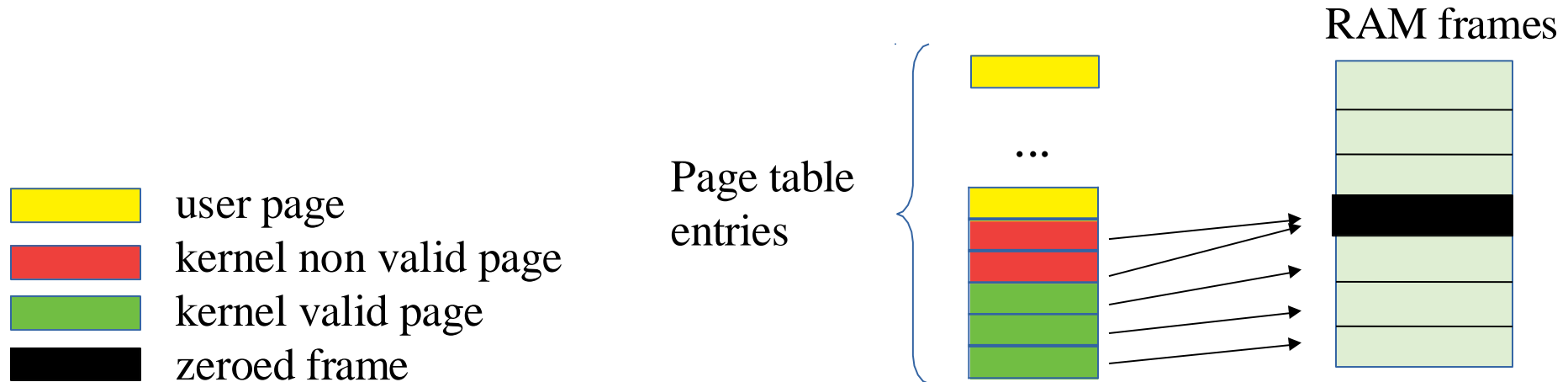
- Kernel 2
  - `swapper_pg_dir`
- Kernel 3
  - `init_level4_pgt`
- Kernel 4/5/6
  - `init_top_pgt` (the **early_top_pgt** one is not the actual kernel page table)

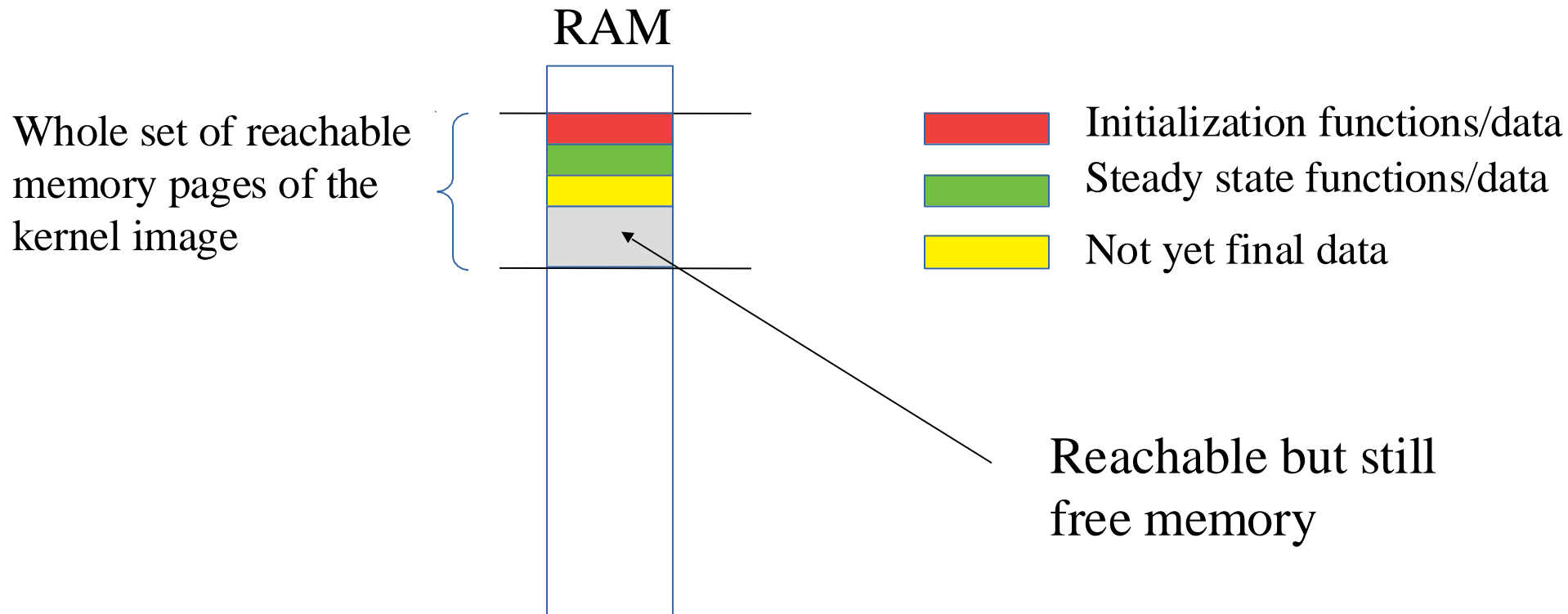# Breaking randomization via Meltdown hardware patches

- The common hardware patch by vendors (e.g Intel) for Meltdown <u>does NOT stall the CPU</u> when performing a user access to a kernel reserved page

- The patch simply <u>passes the default value zero</u> as the target value read by the CPU

- Hence we can only reload from cache the zero-th byte of the PROBE[] array in a meltdown attack (if PTI is not activated at software level)

- However we can execute accesses to different virtual addresses of the kernel when we run in user mode in order to determine if the actual cache-level side effect for the zero-th byte of PROBE[] has happened

- This leads to determine where in the logical address space <u>the base address of the kernel image has been randomized</u>

To protect KASLR we still need software patches

- The kernel side logical pages in the randomization interval can be registered as present in RAM

- Hence the user level speculative access will not stall, and will not lead "non valid" kernel pages to be identified

- All these "non valid" pages can be <u>mapped to the same zeroed physical page</u> in RAM (be carefull anyway to the page size!!)

RAM frames

| | | user page |
| kernel non valid page |
| kernel valid page |
| zeroed frame |

Page table entries

Looking at what we actually loaded in RAM when loading the initial kernel image

RAM

Whole set of reachable memory pages of the kernel image

Initialization functions/data

Steady state functions/data

Not yet final data

Reachable but still free memory

Dealing with initialization stuff

- Initialization stuff <u>becomes completely useless</u> once the kernel has reached steady state behavior

- For an operating system kernel the design always targets eliminating any no longer useful stuff from memory

  - For optimizing resource management (like for, e.g., special purpose kernel configuration)

  - For better coping with the growth of startup complexity

  - For better coping with security aspects (e.g. for eliminating potential gadgets)

- The elimination requires both compile-time and run-time support

  - We need to know what (and where) is stuff only useful at early stage based on compilation choices

  - We need to recover the usage of the corresponding memory pages at run-time

Looking at Linux - hints on the signature of the start_kernel function (as well as others)

…… __init start_kernel(void)

This only lives in memory during kernel boot (or startup)

Recall that the kernel image is not subject to swap out (conventional kernels are always resident in RAM)
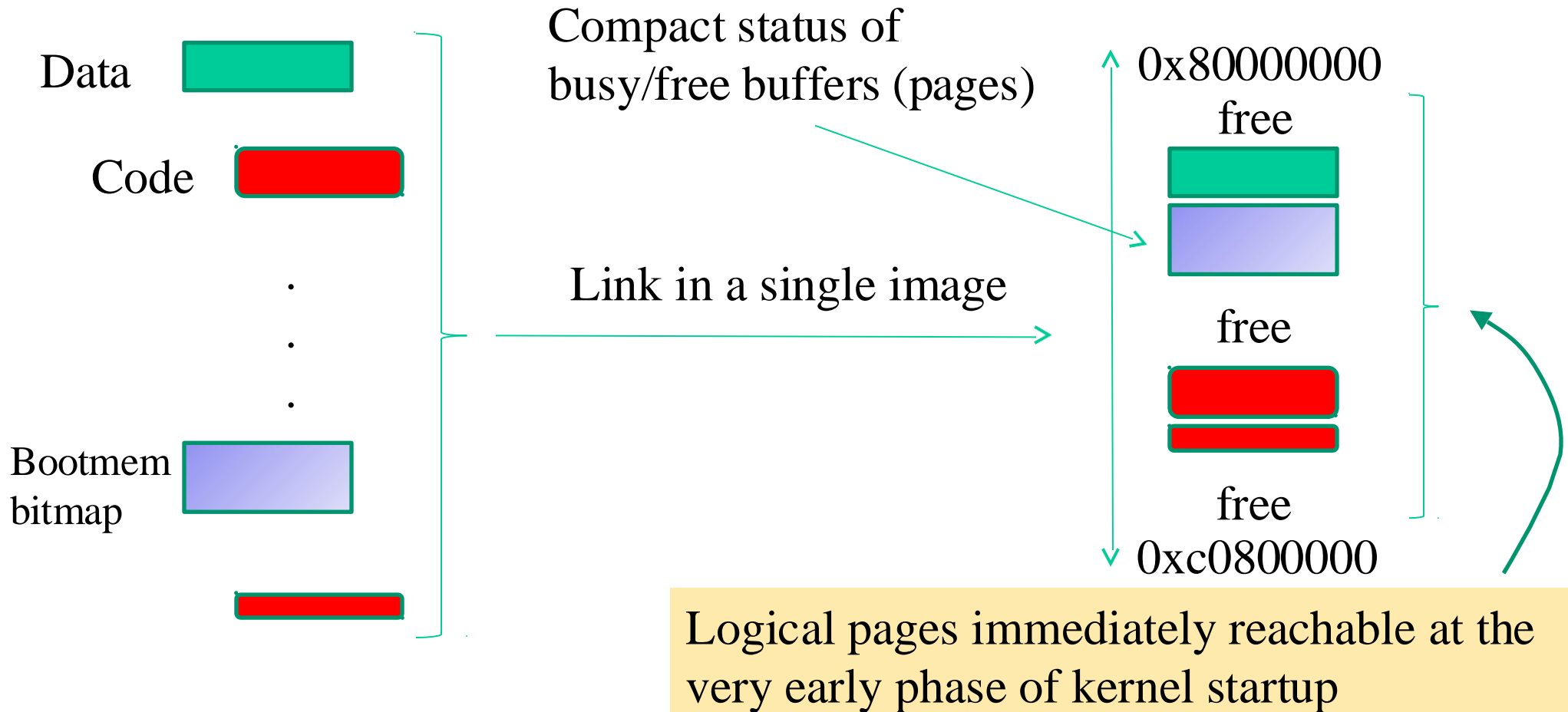
Management of __init functions

- The kernel compile/linking stage locates these functions on specific logical pages (recall what we told about the fixed positioning of specific kernel level stuff in the kernel layout!!)

- These logical pages are identified within a "bootmem" subsystem that is used for managing memory when the kernel is not yet at steady state of its memory management operations

- Essentially the bootmem subsystem keeps a bitmask with one bit indicating whether a given page has been used (at compile time) for specific stuff

… still on bootmem

- The bootmem subsystem also allows for memory allocation upon the very initial phase of the kernel startup

- In fact, the data structures (e.g. the bitmaps) it keeps not only indicate if some page has been used for specific data/code

- They also indicate if some page which is actually reachable at the very early phase of boot is **not used for any stuff**

- These are clearly "free buffers" exploitable upon the very early phase of boot

- Bootmem can be exploited through a specific API

An exemplified picture of bootmem

Data

Code

.
.
.

Bootmem
bitmap

Compact status of
busy/free buffers (pages)

Link in a single image

0x80000000
free

free

free
0xc0800000

Logical pages immediately reachable at the
very early phase of kernel startup

The meaning of ``reachable page''

- The kernel software can access the actual content of the page (in RAM) by simply expressing <u>an address falling into that page</u>

- The point is that <u>the expressed address is typically a virtual one</u> (since kernel code is commonly written using ``pointer'' abstractions)

- So the only way we have to make a virtual address correspond to a given page frame in RAM is to rely on at least a page table that makes the translation (even at the very early stage of kernel boot)

- In early builds of Linux, the initial kernel image had a page table, with a minimum number of pages mapped to RAM, those handled by the bootmem subsystem

How is RAM organized on modern (large scale/parallel) machines?

- In modern chipsets, the CPU-core count continuously increases

- However, it is increasingly difficult to build architectures with a flat-latency memory access (historically referred to as UMA)

- Current machines are typically NUMA

- Each CPU-core has some RAM banks that are close and other that are far

- Generally speaking, each memory bank is associated with a so called NUMA-node

- Modern operating systems are designed to handle NUMA machines (hence UMA as a special case)

Looking at the Linux NUMA setup via Operating System facilities

- A very simple way is the `numactl` command
- It allows to discover

  ✓ How many NUMA nodes are present

  ✓ What are the nodes close/far to/from any CPU-core

  ✓ What is the actual distance of the nodes (from the CPU-cores)

  Let's see a few 'live' examples …….

Bootmem vs Memblock

- In more recent versions of OS kernels the bootmem architecture has been enriched

- It allows for keeping track of free/busy frames with a per-NUMA node granularity

- The newer architecture is called "memblock" in Linux

- An additional logic is inserted for setting up the memblock system to indicate how many NUMA nodes we have

- The API for managing memory in memblock has been slightly changed with respect to traditional bootmem

- However the essence of the operations we can do is the same
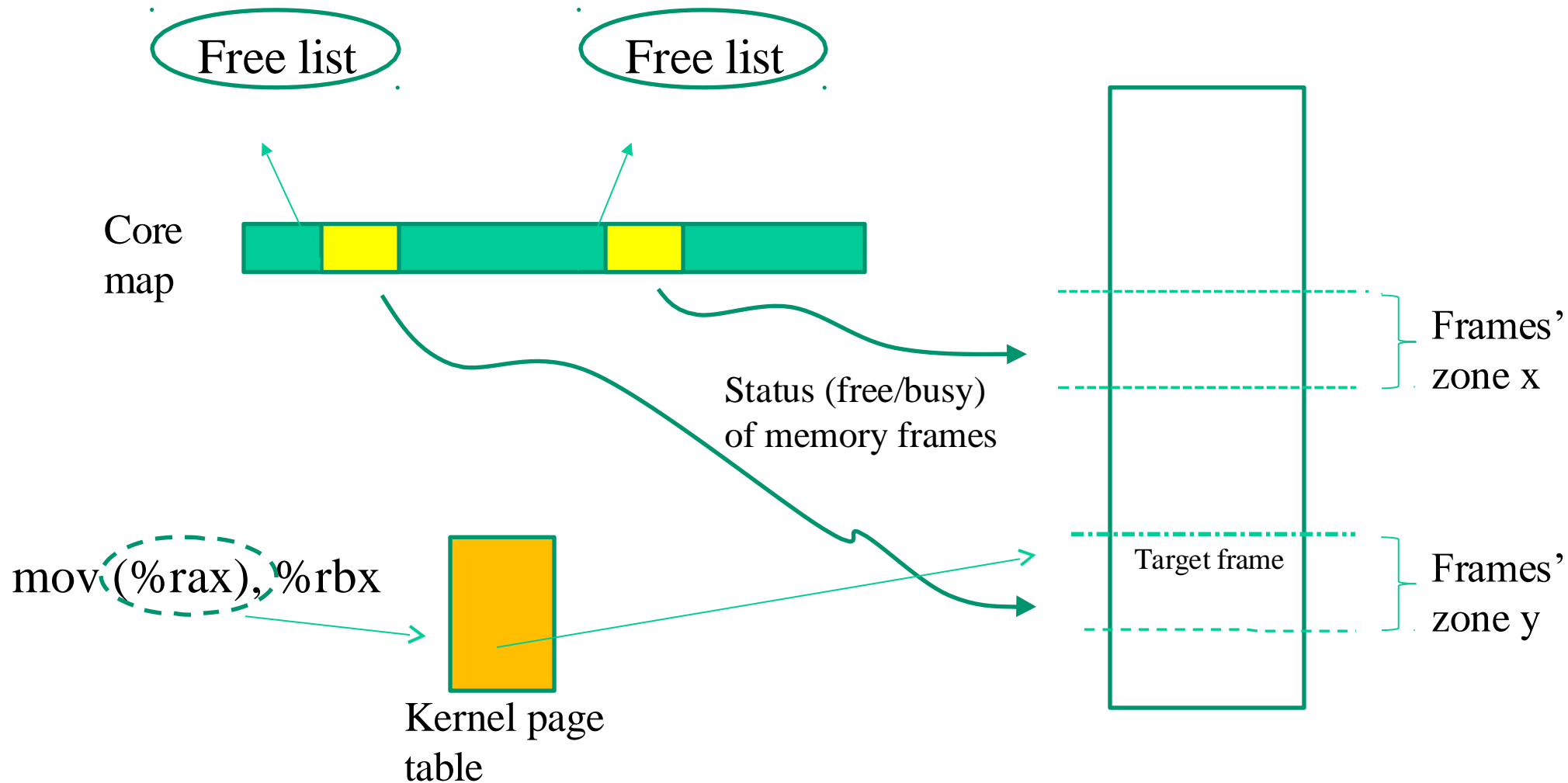
Bootmem vs Memblock allocation API examples

- in bootmem an allocation deals with "low pages"

  ➢ `alloc_bootmem_lowpages()` - this returns the virtual address of the allocated memory

- in memblock the classical "low pages" allocators have been encapsulated into a slightly different API functions

  ➢ `memblock_phys_alloc*()` - these functions return the physical address of the allocated memory

  ➢ `memblock_alloc*()` - these functions return the virtual address of the allocated memory

Actual kernel data structures for managing memory

- Kernel (identity) Page table
  - ➤ This is a kind of 'ancestral' page table (all the others are somehow derived from this one)
  - ➤ 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

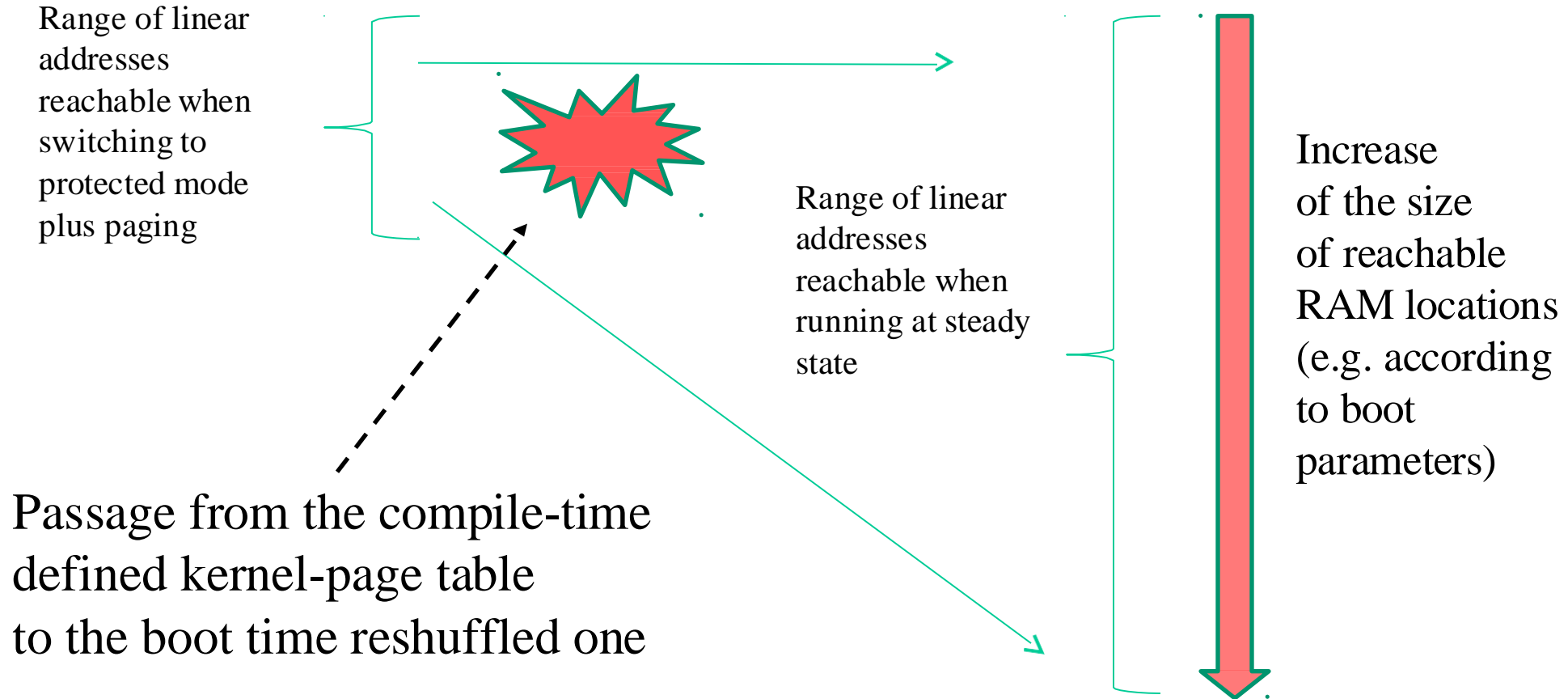None of them is already finalized when we startup the kernel

A scheme

Free list

Free list

Core map

Status (free/busy) of memory frames

Frames' zone x

mov (%rax), %rbx

Kernel page table

Target frame

Frames' zone y

# Objectives of the kernel (identity) page table setup

- These are basically two:

  - ✓ Allowing the kernel software to use virtual addresses while executing (either at startup or at steady state)

  - ✓ Allowing the kernel software (and consequently the application software) to reach (in read and/or write mode) the maximum admissible (for the specific machine) or available RAM storage

- The finalized shape of the kernel page table is therefore typically not setup into the original image of the kernel loaded in memory

  - given that the available RAM to drive can be parameterized

  - given that we may have randomization

A classical scheme for Linux/x86 protected mode (i386)

Range of linear addresses reachable when switching to protected mode plus paging

Range of linear addresses reachable when running at steady state

Increase of the size of reachable RAM locations (e.g. according to boot parameters)

Passage from the compile-time defined kernel-page table to the boot time reshuffled one

Directly mapped memory pages

- They are kernel level pages whose mapping onto physical memory (frames) is based on a simple shift between virtual and physical addresses

  ✓ $PA = \Omega(VA)$ where Ω is (typically) a simple function subtracting a predetermined and known constant value to VA

- Not all the kernel level virtual pages are directly mapped

Kernel level logical pages

Directly mapped
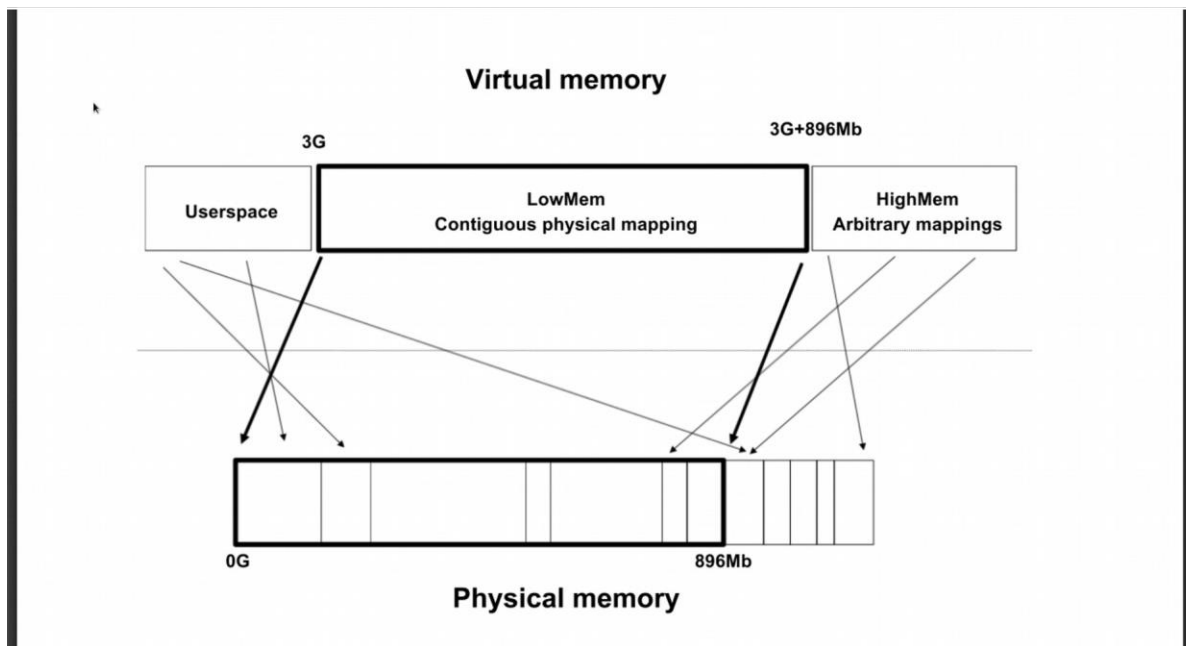
Non-directly mapped

Page frames
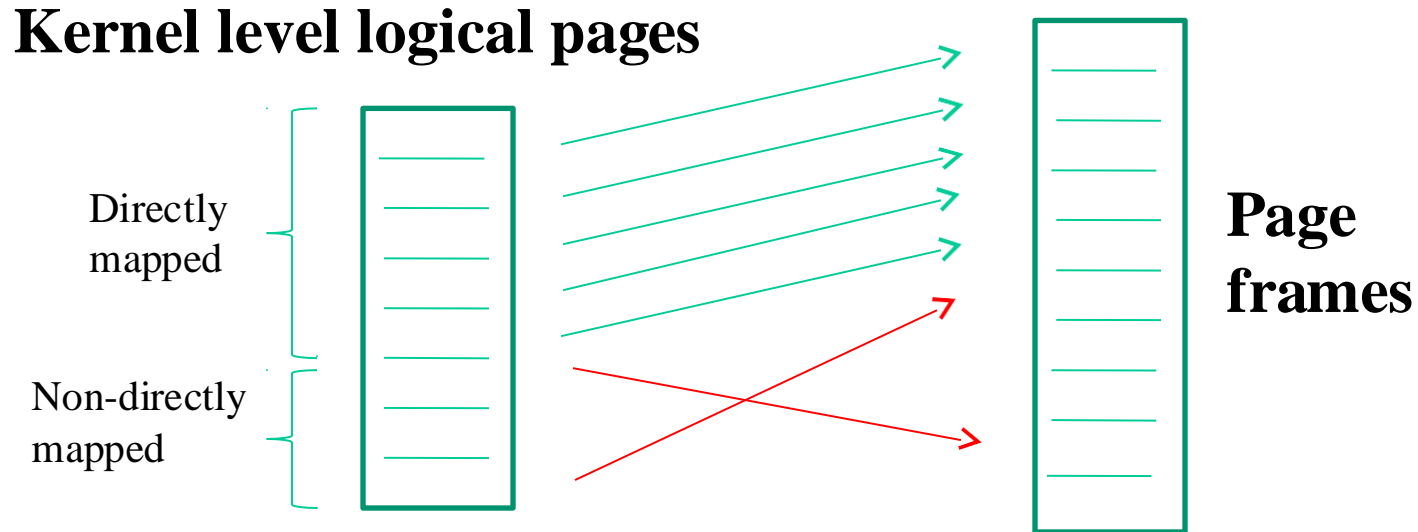
Page mapping vs ZONEs

- It is typical for an operating system kernel to organize physical memory into ZONEs

- The ZONE determines the type of usage we need to do with pages

- The more commonly known zones are

  - DMA

  - NORMAL

  - HIGH

- DMA is used for reserving memory to specific device operations

- NORMAL is used for directly mapped pages from the kernel

- HIGH is used for non-directly mapped pages from the kernel and else (e.g. user stuff)

  - Useful when, e.g., physical memory is close to or greater than virtual memory

  - Useful for flexibility of memory mapping

A simple case of Linux and x86 protected mode

```
ZONE_DMA              < 16 MB      ISA DMA capable memory

ZONE_NORMAL        16-896 MB   direct mapped by the kernel

ZONE_HIGHMEM         > 896 MB   only page cache and user
```
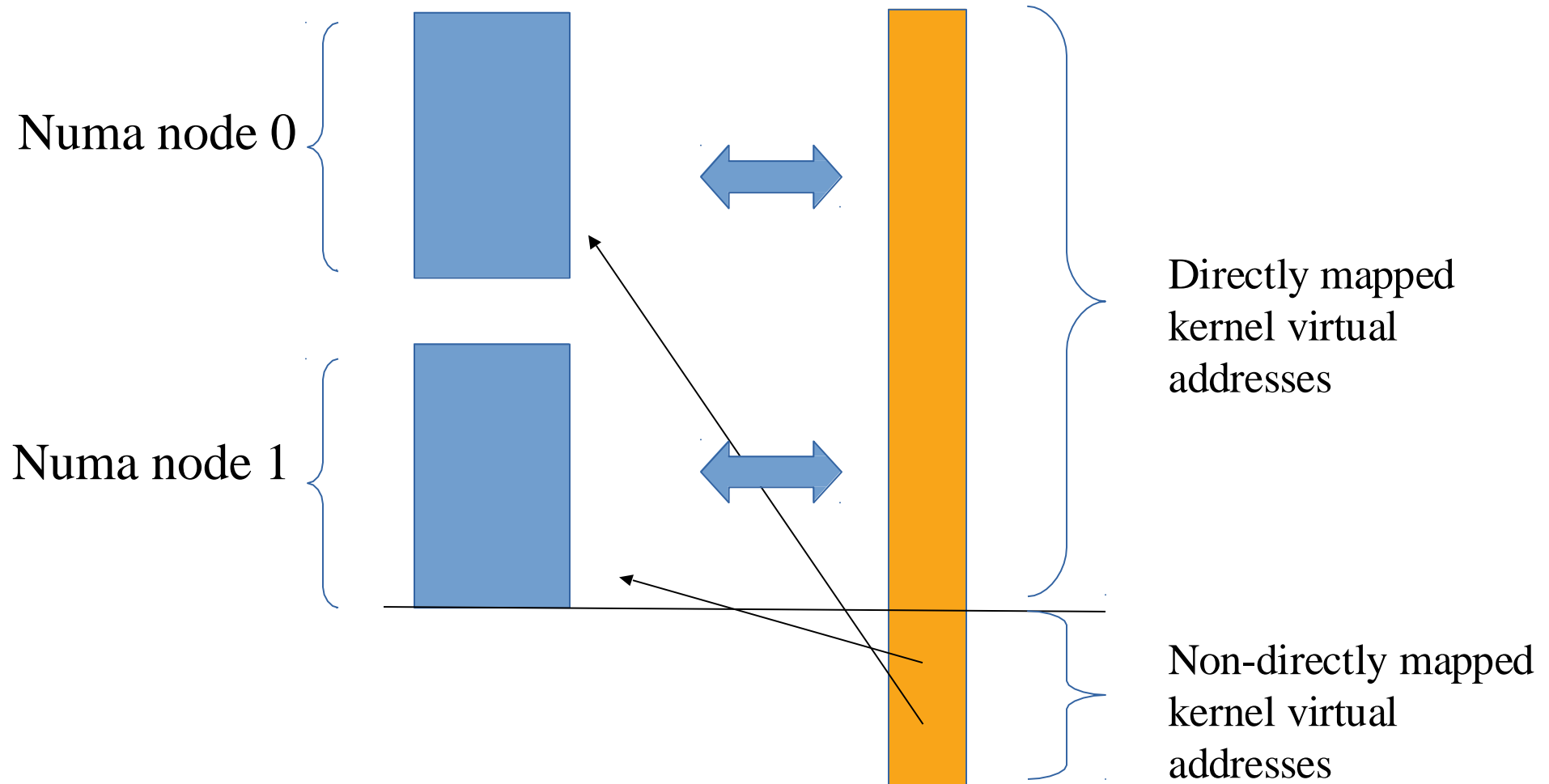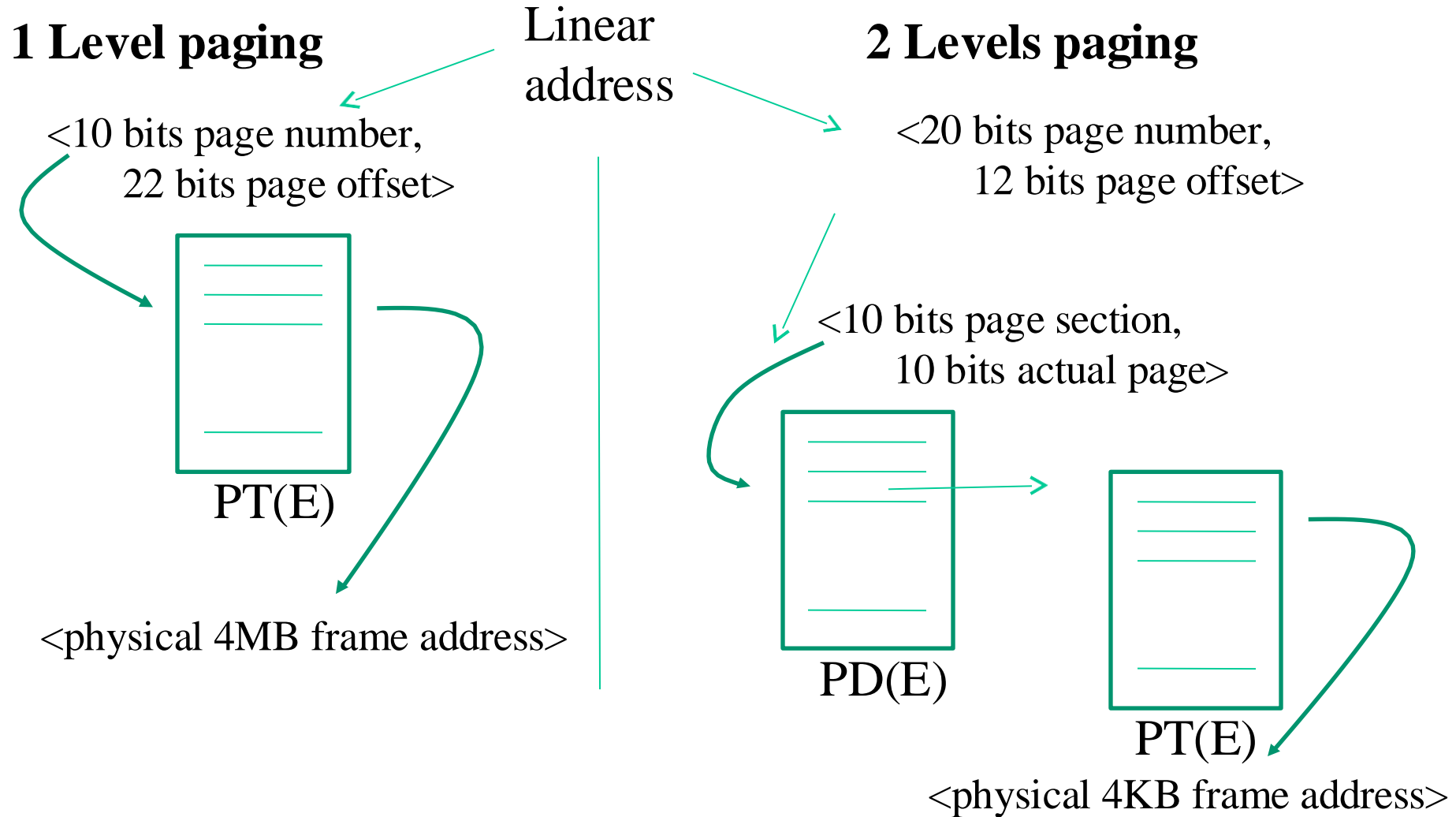
A scheme ⟶

The case of Linux and x86-64 (long mode)

- The kernel takes all the RAM for direct mapping

- The kernel can also take the whole RAM memory for non-directly mapped stuff

- This capability simply derives from the extremely enlarged possibility to address logical and physical memory with an x86-64 processor

- As we will see we can in fact use 2^48 bytes in the logical address space

- ZONEs are no longer relevant, but non direct mapping still stands there

The scheme for Linux/x86-64 (long mode)

Page table structure in x86 protected mode (i386)

1 Level paging

Linear address

2 Levels paging

<10 bits page number,
22 bits page offset>

<20 bits page number,
12 bits page offset>

PT(E)

<10 bits page section,
10 bits actual page>

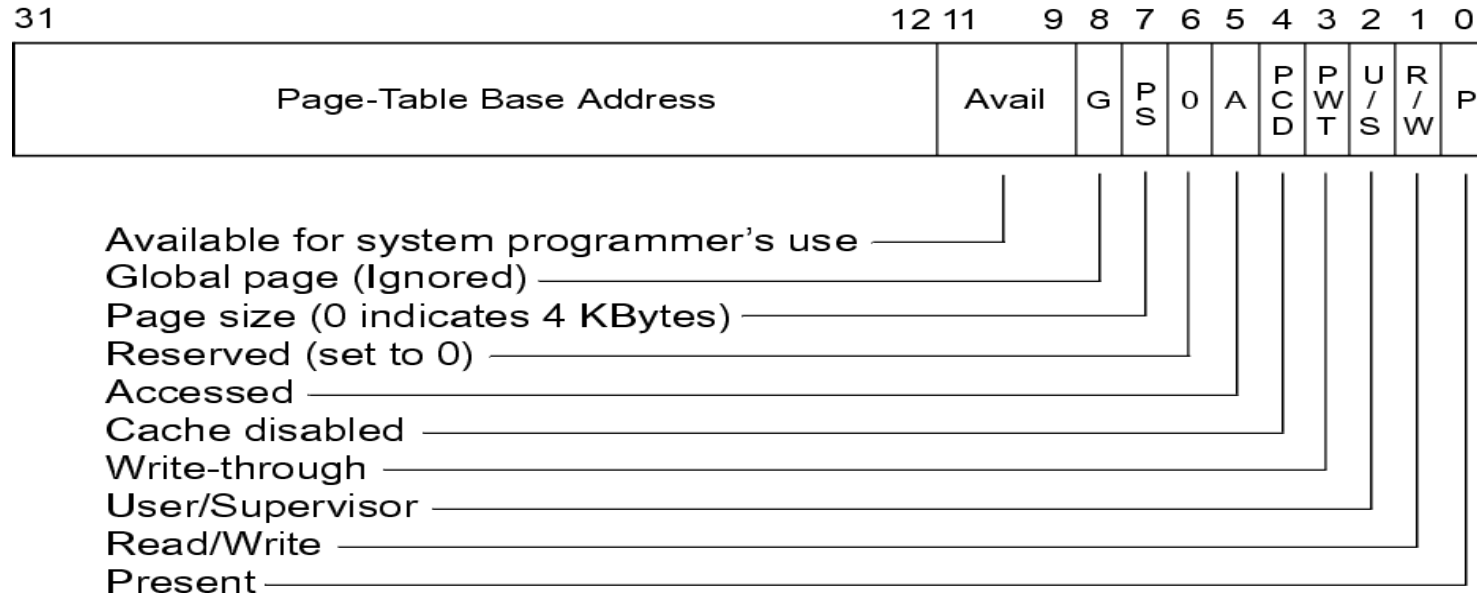<physical 4MB frame address>

PD(E)

PT(E)

Page table entries

- Each page table entry is made of 32 bits (and there are 1024 in each page table)

- Each (part of) the i386 page table needs to be allocated in memory aligned to the 4K address

- If we are working at one or two levels is determined by the control bits kept into the first level page table

- The maximum amount of logical memory we can access is 4GB

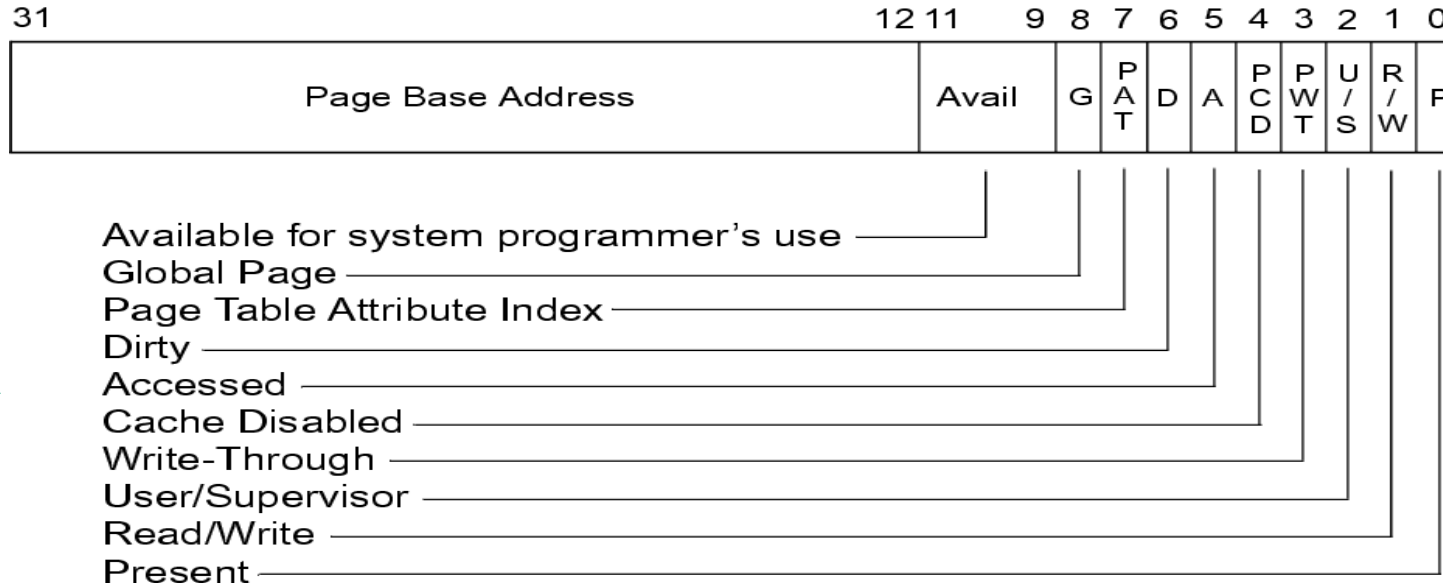- The maximum amount of physical memory we can access is 4GB

i386 PDE entries

Page-Directory Entry (4-KByte Page Table)

31 12	11 9	8	7	6	5	4	3	2	1	0
Page-Table Base Address	Avail	G	P S	0	A	P C D	P W T	U / S	R / W	P

Available for system programmer's use
Global page (Ignored)
Page size (0 indicates 4 KBytes)
Reserved (set to 0)
Accessed
Cache disabled
Write-through
User/Supervisor
Read/Write
Present

Nothing tells whether we can fetch (hence execute) from there

i386 PTE entries

Page-Table Entry (4-KByte Page)

31			12	11		9	8	7	6	5	4	3	2	1	0
Page Base Address				Avail			G	PAT	D	A	PCD	PWT	U/S	R/W	P

Available for system programmer's use ——————
Global Page ——————
Page Table Attribute Index ——————
Dirty ——————
Accessed ——————
Cache Disabled ——————
Write-Through ——————
User/Supervisor ——————
Read/Write ——————
Present ——————

Nothing tells whether we can fetch (hence execute) from there
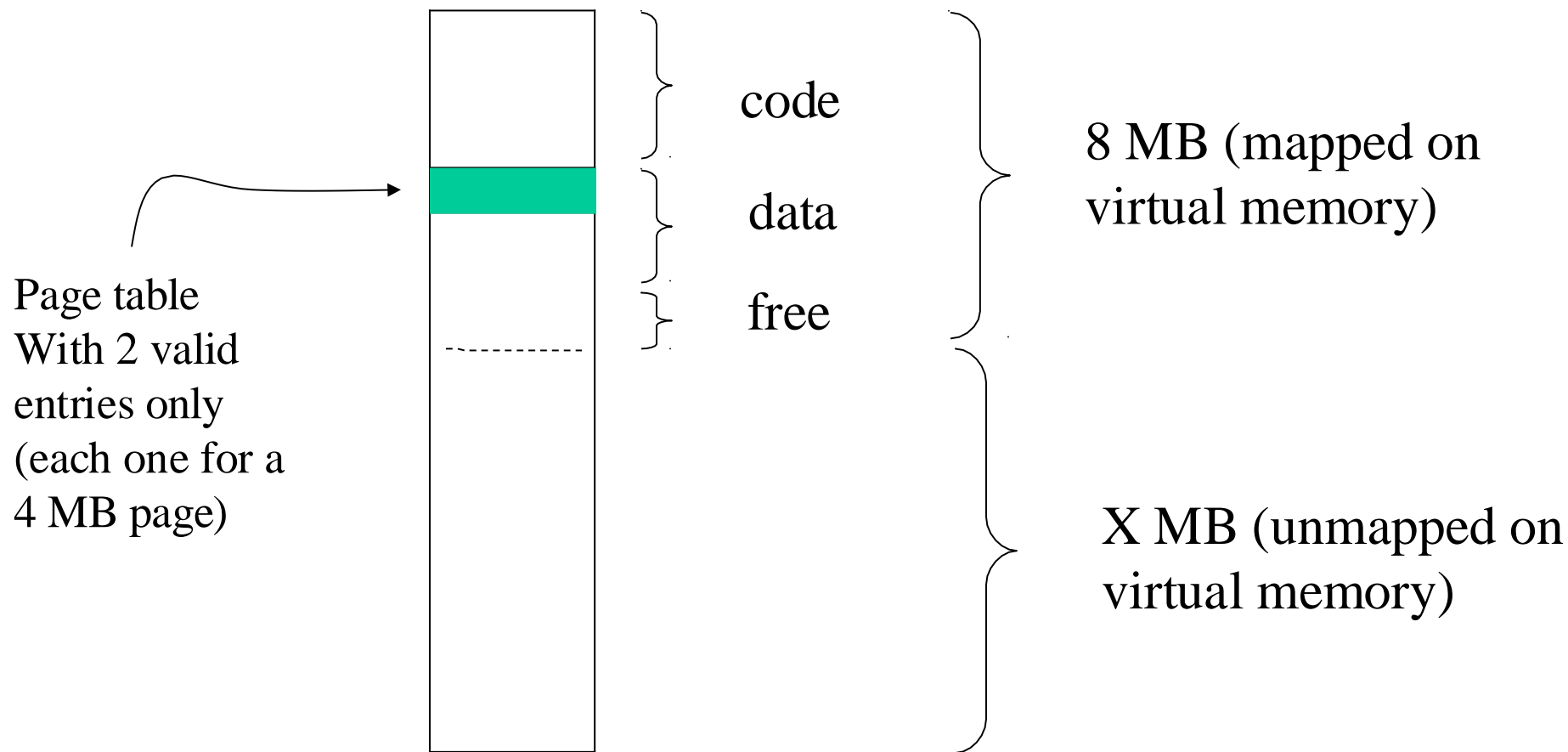
Field semantics

- **Present**: indicates whether the page or the pointed page table is loaded in physical memory. This flag is not set by firmware (rather by the kernel)

- **Read/Write**: define the access privilege for a given page or a set of pages (as for PDE) . Zero means read only access

- **User/Supervisor**: defines the privilege level for the page or for the group of pages (as for PDE). Zero means supervisor privilege

- **Write Through**: indicates the caching policy for the page or the set of pages (as for PDE). Zero means write-back, non-zero means write-through

- **Cache Disabled**: indicates whether caching is enabled or disabled for a page or a group of pages. Non-zero value means disabled caching (as for the case of memory mapped I/O)

- **Accessed**: indicates whether the page or the set of pages has been accessed. This is a sticky flag **(no reset by firmware)**. Reset is controlled via software

- **Dirty**: indicates whether the page has been write-accessed. This is also a sticky flag

- **Page Size (PDE only)**: if set indicates **4 MB paging otherwise 4 KB paging**

- **Page Table Attribute Index: ….. Do not care ……**

- **Page Global (PTE only)**: defines the caching policy for TLB entries. Non-zero means that **the corresponding TLB entry does not require reset upon loading a new value into the page table pointer CR3**

Page table setup at boot in Linux - the i386/kernel2.4 very didactical example

- Upon kernel startup addressing relies on a simple single level paging mechanism that only maps 2 pages (each of 4 MB) up to 8 MB physical addresses

- The physical address of the setup page table is kept within the CR3 register

- The page table is updated in order to enable reaching up to 1 GB of memory via kernel level addresses

- Recall that on i386 processors Linux has

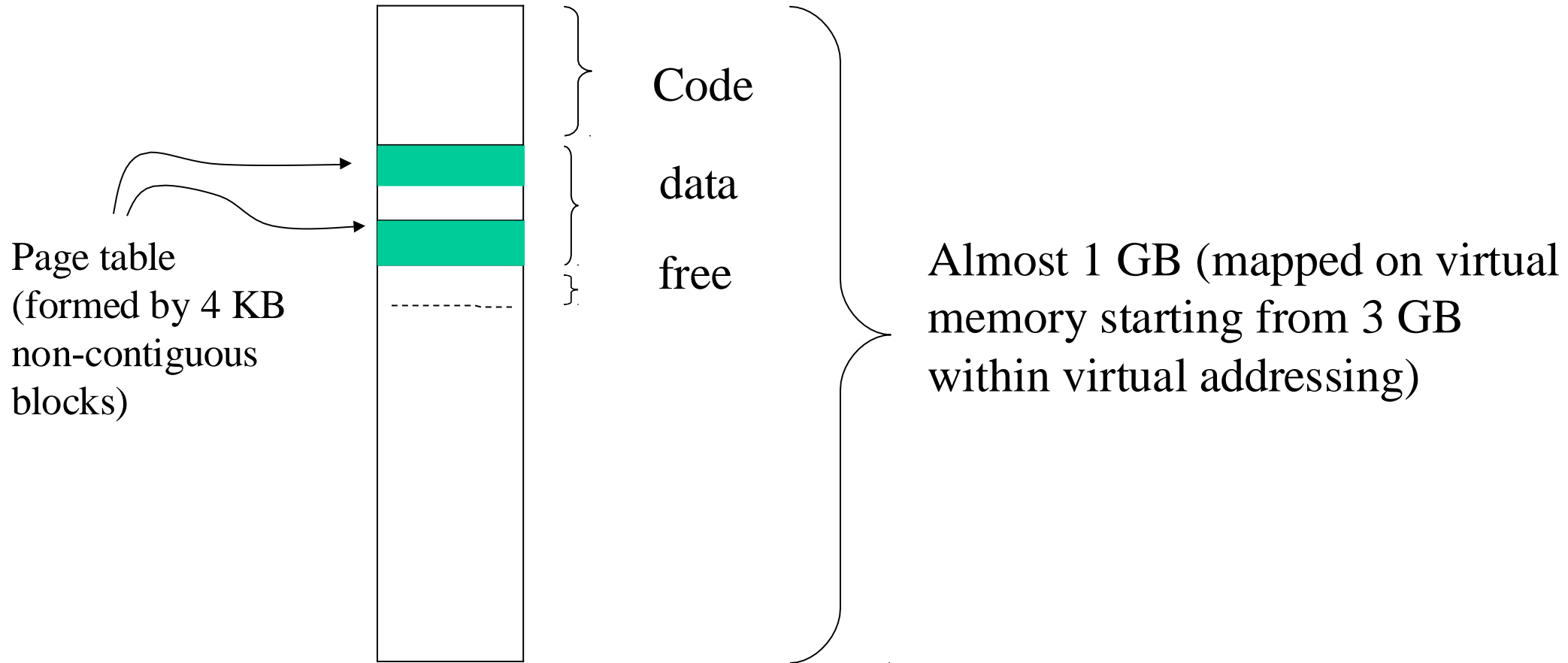   – [0, 3GB) for user stuff

    – [3GB, 4GB] for kernel stuff

i386/kernel2.4 memory layout at kernel startup

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

code

data

free

8 MB (mapped on
virtual memory)

X MB (unmapped on
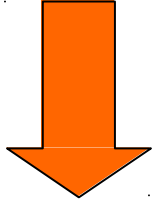virtual memory)

Issues to be tackled

1. We need to reach the correct granularity for paging (4KB rather than 4MB)

2. We need to span logical to physical address across the whole 1GB of kernel-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 yet at steady state)

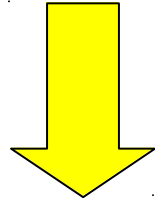# i386/kernel2.4 memory layout at steady state

Code

data

free

Page table
(formed by 4 KB
non-contiguous
blocks)

Almost 1 GB (mapped on virtual
memory starting from 3 GB
within virtual addressing)

The passage through low level "pages"

Load undersized page table
(kernel page size not finalized: 4MB)
- 4 KB (1K entry)

Finalize kernel handled
page size (4KB)

Expand page table via
bootmem low pages
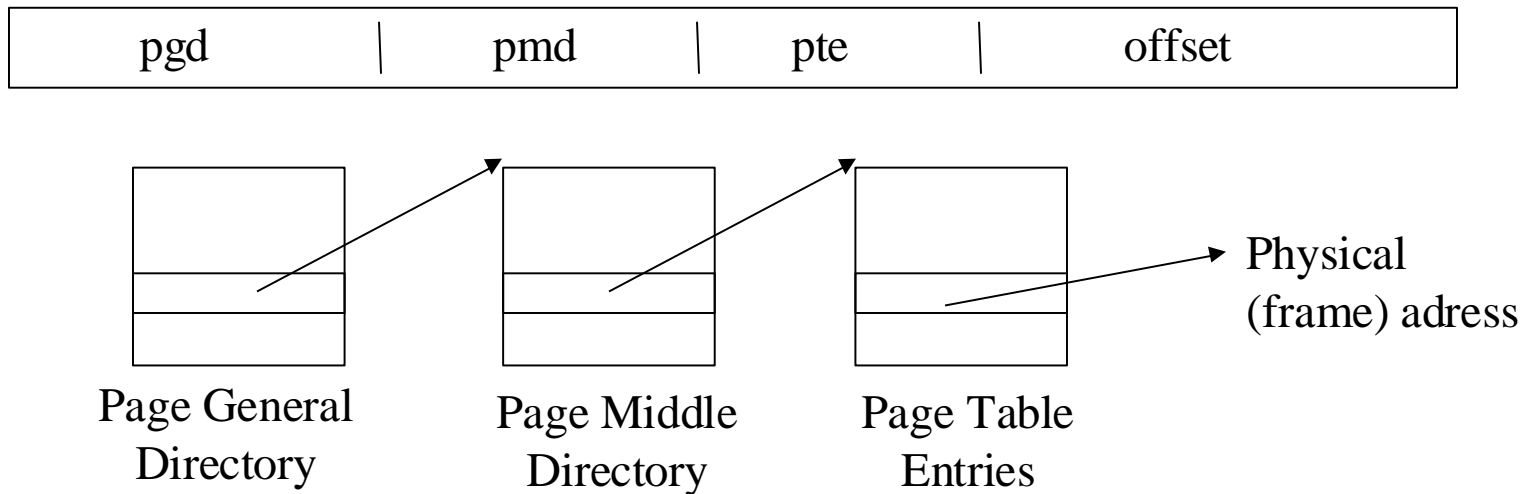(not marked in the page table)
- compile time identification
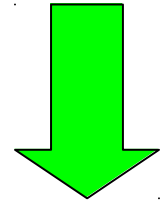
Kernel boot

i386/kernel2.4 paging data structures

- Linux virtual addresses since kernel 2.4 exhibit (at least) 3 indirection levels

pgd	pmd	pte	offset

Page General Directory   Page Middle Directory   Page Table Entries   Physical (frame) adress

- On i386 machines, paging is supported limitedly to 2 levels (`pde`, page directory entry – `pte`, page table entry)

  ➤ `pgd` Linux maps to i386 `pde`

  ➤ `pte` Linux maps to i386 `pte`

Some useful macros

- The following macros define the size of the page tables blocks (they can be found in the file `include/asm-i386/pgtable-2level.h`)

    - ➤ `#define PTRS_PER_PGD    1024`

    - ➤ `#define PTRS_PER_PMD    1`

    - ➤ `#define PTRS_PER_PTE    1024`

- the value1 for `PTRS_PER_PMD` is used **to simulate the existence of the intermediate level** such in a way to keep the 3-level oriented software structure to be compliant with the 2-level architectural support

Some useful data structures

- A core structure is represented by the symbol `swapper_pg_dir` which is defined within the file `arch/i386/kernel/head.S`

- This symbol expresses the virtual memory address of the **PGD (PDE)** portion of the kernel page table

- This value 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 starting from the initial PGD address

- **The C types for the definition of the content of the page table entries on i386** are defined in `include/asm-i386/page.h`

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

Debugging

- The redefinition of different structured types, which are identical in size and equal to an `unsigned long,` is done for debugging purposes

- Specifically, in C technology, **different aliases for the same type are considered as identical types**

- For instance, if we define

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

the compiler enables assignments such as `x=y` and `y=x`

- Hence, there is the need for defining different structured types which simulate the base types that would otherwise give rise to compiler equivalent aliases

Bit-masks

- in `include/asm-i386/pgtable.h` there exist some macros defining the positioning of control bits within the entries of the PDE or PTE

- There also exist the following macros for masking and setting those bits

  - ➢`#define _PAGE_PRESENT    0x001`
  - ➢`#define _PAGE_RW         0x002`
  - ➢`#define _PAGE_USER       0x004`

     ...
  - ➢`#define _PAGE_ACCESSED   0x020`
  - ➢`#define _PAGE_DIRTY      0x040 /* proper of PTE */`

- These are all machine dependent macros

An example

```
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 */
} ;
```

i386/kernel2.4 initialization algorithm

- we start by the PGD entry which maps the address 3 GB, namely the entry numbered 768

- cyclically

  1. We determine the virtual address to be memory mapped (this is kept within the `vaddr` variable)

  2. One page for the PTE table gets allocated which is used for mapping 4 MB of virtual addresses

  3. The table entries are populated

  4. The virtual address to be mapped gets updated by adding 4 MB

  5. We jump to step 1 unless no more virtual addresses or no more physical memory needs to be dealt with (the ending condition is recorded by the variable `end`)

pagetable_init()

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

    }
}
```

Important note

- **<u>The final PDE buffer coincides with the initial page table that maps</u>** 4 MB pages

- 4KB paging gets activated upon filling the entry of the PDE table (since the Page Size bit gets updated)

- For this reason the PDE entry is set only after having populated the corresponding PTE table to be pointed

- **<u>Otherwise memory mapping would be lost upon any TLB miss</u>**

What about more recent kernels (e.g. kernel 5/6)?

- The /arch /x86/include /asm /pgtable_64_types.h header file reports the types for the entries of the page tables

- In this .h file we can see how Linux is already configured for a 5-level paging scheme

- As we will shortly discuss, we currently use up to 4 levels when setting up and running the Linux kernel on an x86 machine operating in long mode

- The fifth level will be actually exploited when the hardware will enable it

Direct memory-mapping macros

- We know that directly mapped logical memory (e.g. the one used for bootmem) is located in physical memory by replying on a fixed offset

- The `__pa()` (physical address) macro allows retrieving the virtual address that corresponds to the physical address passed as input parameter

- **This mapping rule is anyhow not valid for all kernel level addresses, since some of them might be non-directly mapped**

- The `__va()` (virtual address) macro allows performing the dual mapping between physical and virtual addresses – still for directly mapped memory

Relations with trap/interrupt events

- Upon a TLB miss, firmware accesses the page table

- The first checked bit is typically `_PAGE_PRESENT`

- If this bit is zero, a page fault occurs which gives rise to a trap (with a given displacement within the trap/interrupt table)

- Hence the instruction that gave rise to the trap can get finally re-executed

- **<u>Re-execution might give rise to additional traps, depending on firmware checks on the page table</u>**

- As an example, the attempt to access a read only page in write mode will give rise to a trap (which triggers the **<u>segmentation fault handler</u>**)

An additional baseline example -
run time detection of current page size (still for i386)

```
#include <kernel.h>

#define MASK 1<<7

unsigned long addr = 3<<30; // fixing a reference on the
                 // kernel boundary

asmlinkage int sys_page_size(){

  //addr = (unsigned long)sys_page_size; // moving the reference
   return(swapper_pg_dir[(int)((unsigned long)addr>>22)]&MASK?
      4<<20:4<<10);
}
```

PAE - Physical address extension

- increase of the bits used for physical addressing (e.g. Intel Pentium Pro)
- provides 36 bits for physical addressing
- we can drive up to 64 GB of RAM memory
- <u>paging operates at 3 levels (instead of 2)</u>
- the traditional page tables get modified by extending the entries at
  64-bits and reducing their number by a half (hence we can support
  ¼ of the address space)
- an additional top level table gets included called "page directory
  pointer table" which entails 4 entries, pointed by CR3
- CR4 indicates whether PAE mode is activated or not (which is done
  via bit 5 – PAE-bit)
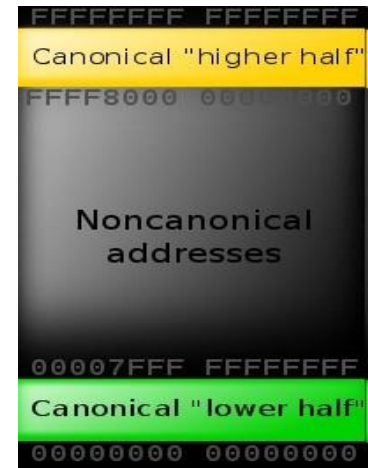
x86-64 architectures

- They extend the PAE scheme via a so called "long addressing mode"

- Theoretically they allow addressing 2^64 bytes of logical memory

- In actual implementations we reach up to 2^48 canonical form addresses (lower/ upper half within a total address space of 2^48)

- The total allows addressing to span over 256 TB

- Not all operating systems allow exploiting the whole range up to 256 TB of logical/physical memory

- LINUX currently allows 128 TB for logical addressing of individual processes and 64 TB for physical addressing
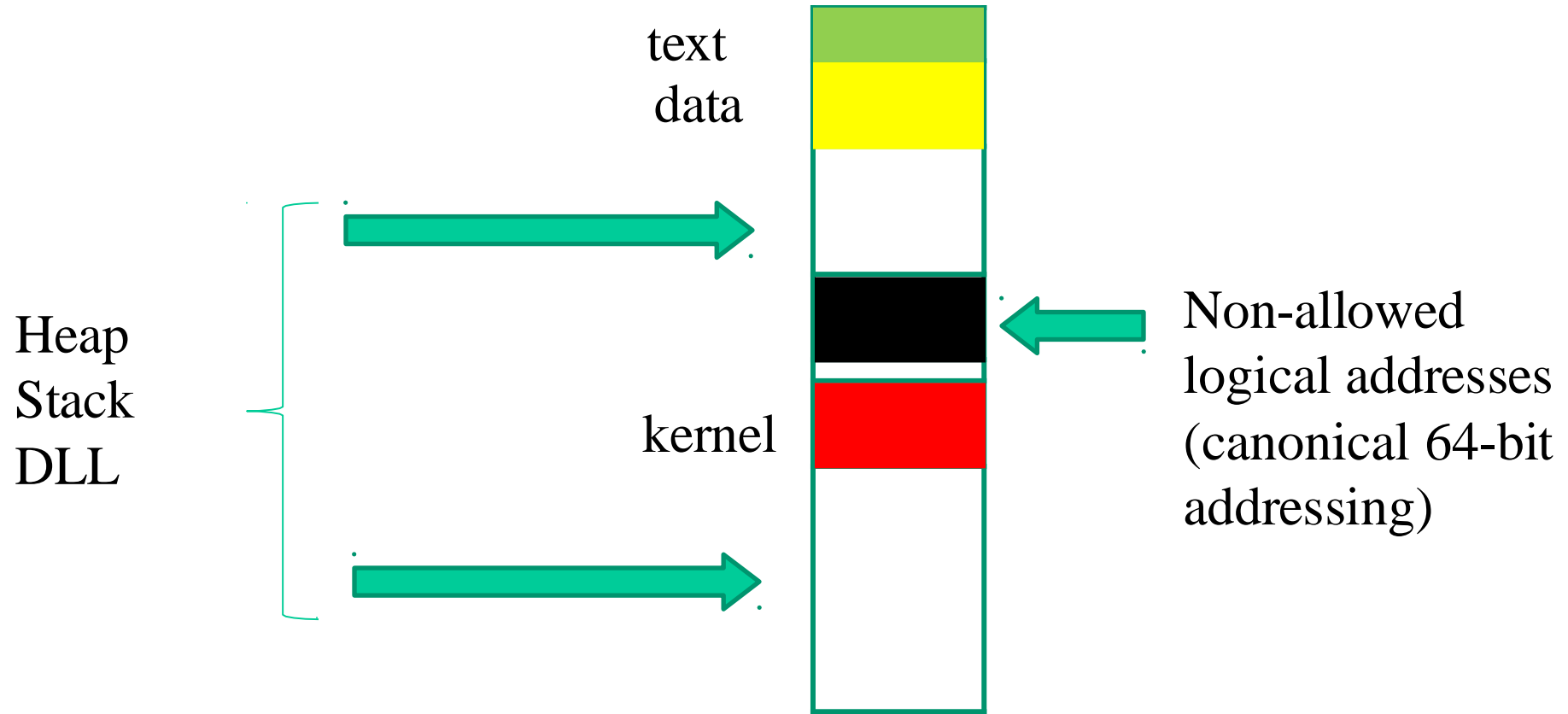
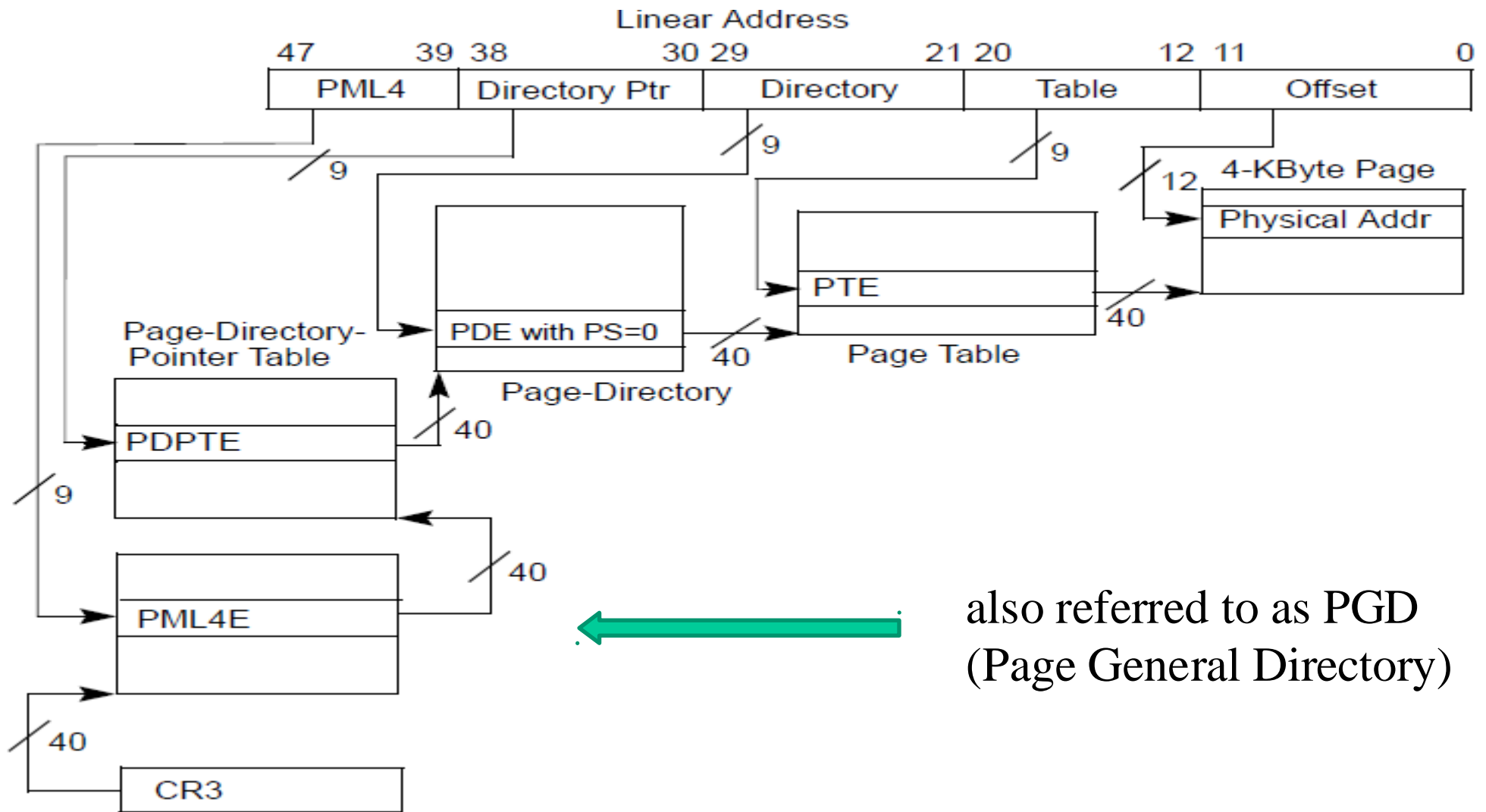# Addressing scheme

64-bit

48 out of 64-bit

Linux address space on x86-64 processors

text
data

Heap
Stack
DLL

kernel

Non-allowed
logical addresses
(canonical 64-bit
addressing)

48-bit addressing - page tables

- Page directory pointer has been expanded from 4 to 512 entries

- An additional paging level has been added thus reaching 4 levels, this is called "Page-Map level"

- Each Page-Map level table has 512 entries

- Hence, we get 512^4 pages of size 4 KB that are addressable (namely, a total of 256 TB)

Linear Address

47	39	38	30	29	21	20	12	11	0
PML4		Directory Ptr		Directory		Table		Offset	

4-KByte Page

Physical Addr

PTE

Page Table

PDE with PS=0

Page-Directory

Page-Directory-Pointer Table

PDPTE

PML4E

CR3

also referred to as PGD (Page General Directory)

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

63 62 61 60 59 58 57 56 55 54 53 52 51	M[1]	M-1 ... 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
Reserved[2]		Address of PML4 table	Ignored					PCD	PWT	Ign.			**CR3**
XD	Ignored / Rsvd.	Address of page-directory-pointer table	Ign.	Rsvd	Ign		A	PCD	PWT	U/S	R/W	1	**PML4E: present**
Ignored												0	**PML4E: not present**
XD	Ignored / Rsvd.	Address of 1GB page frame / Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S R/W	1	**PDPTE: 1GB page**
XD	Ignored / Rsvd.	Address of page directory	Ign.	0	Ign		A	PCD	PWT	U/S	R/W	1	**PDPTE: page directory**
Ignored												0	**PDTPE: not present**
XD	Ignored / Rsvd.	Address of 2MB page frame / Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S R/W	1	**PDE: 2MB page**
XD	Ignored / Rsvd.	Address of page table	Ign.	0	Ign		A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored												0	**PDE: not present**
XD	Ignored / Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored												0	**PTE: not present**

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

Table 4-19. Format of an IA-32e Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
(M−1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Direct vs non-direct page mapping

- In long mode x86 processors allow one entry of the PML4 to be associated with 2^27 frames

- This amounts to 2^29 KB = 2^9 GB = 512 GB

- <u>Clearly, we have plenty of room in virtual addressing for directly mapping all the available RAM into kernel pages on most common chipsets</u>

- This is the typical approach taken by Linux, where we directly map all the RAM memory

- However, we also remap the same RAM memory in non-direct manner whenever required
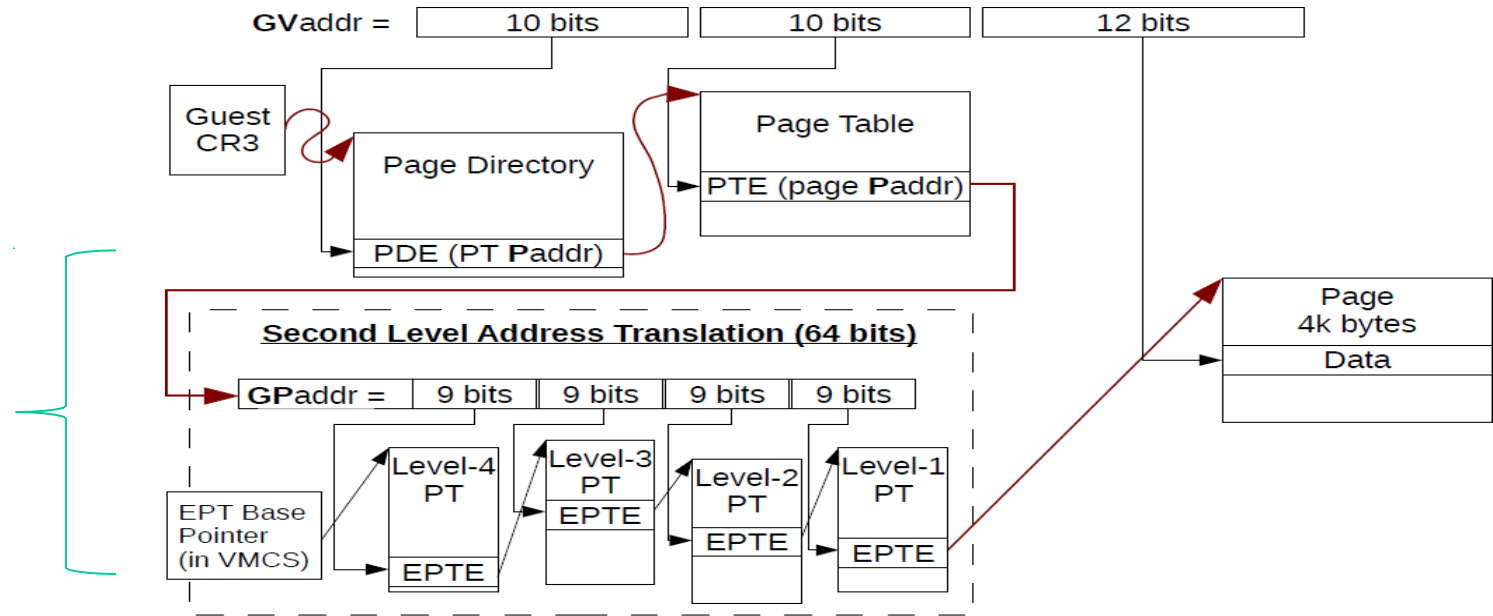
Huge pages

- Ideally x86-64 processors support them starting from PDPT

- Linux typically offers the support for huge pages pointed to by the PDE (page size 512*4KB)

- See: `/proc/meminfo` and `/proc/sys/vm/nr_hugepages`

- These can be "mmaped" via file descriptors and/or mmap parameters (e.g. MAP_HUGETLB flag)

- They can also be requested via the `madvise(void*, size_t, int)` system call (with MADV_HUGEPAGE flag)

Hardware supported "virtual memory" virtualization

- Intel Extended Page Tables (EPT)

- AMD Nested Page Tables (NPT)

- A scheme:

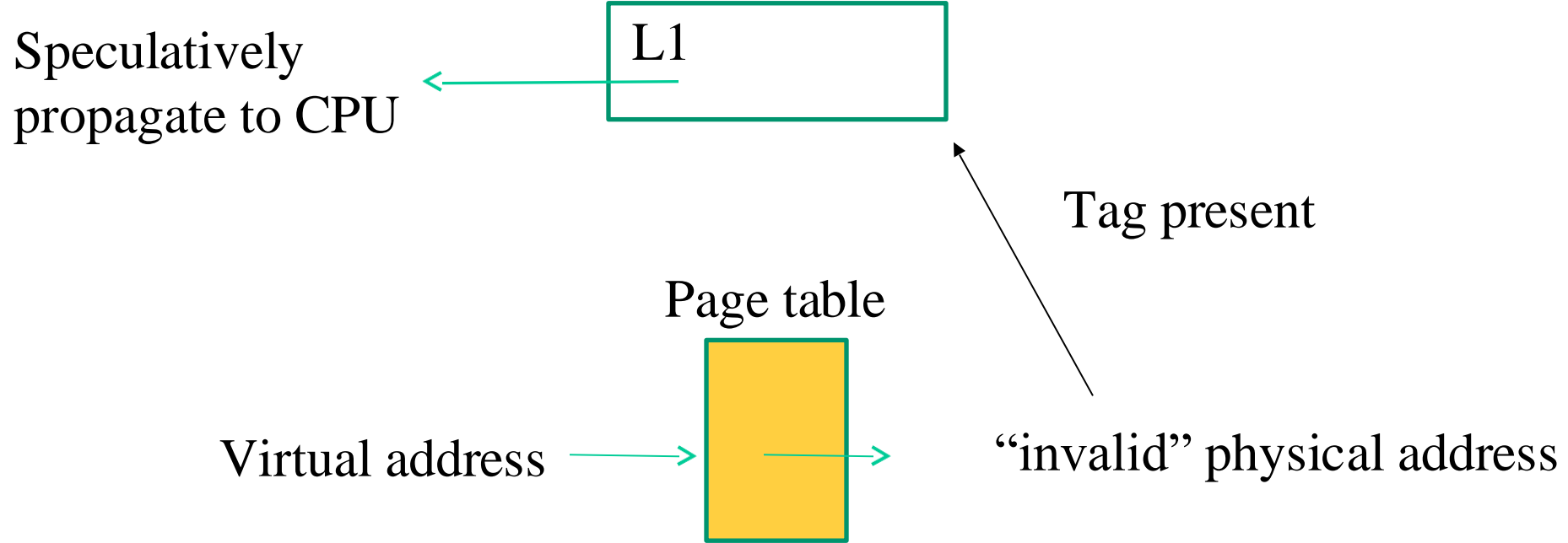Keeps track of the physical memory location of the page frames used for activated VM

GVaddr = | 10 bits | 10 bits | 12 bits |

Guest CR3

Page Directory
- PDE (PT Paddr)

Page Table
- PTE (page Paddr)

Page 4k bytes
- Data

Second Level Address Translation (64 bits)

GPaddr = | 9 bits | 9 bits | 9 bits | 9 bits |

EPT Base Pointer (in VMCS)

Level-4 PT
- EPTE

Level-3 PT
- EPTE

Level-2 PT
- EPTE

Level-1 PT
- EPTE

Back to speculation in the hardware

- From Meldown we already know that a page table entry plays a central role in hardware side effects with speculative execution

- The page table entry provides the physical address of some "non-accessible" byte, which is still accessible in speculative mode

- This byte can flow into speculative incarnations of registers and can be used for cache side effects

- **….. but, what about a page table entry with "presence bit" not set???**

- **….. is there any speculative action that is still performed by the hardware with the content of that page table entry?**

The L1 Terminal Fault (L1TF) attack

- It is based on the exploitation of data cached into L1

- More in detail:

  - A page table entry with presence bit set to 0 propagates the value of the target physical memory location (the TAG) upon loads if that memory location is already cached into L1

  - If we use the value of that memory location as an index (Meltdown style) we can grub it via side effects on cache latency

- Overall, we can be able to indirectly read the content of any physical memory location if the same location has already been read, e.g., in the context of another process on the same CPU-core

- Affected CPUs: Intel ATOM, Intel Xeon PHI …

The scheme

Speculatively
propagate to CPU

L1

Tag present

Page table

Virtual address

"invalid" physical address

L1TF big issues

- To exploit L1TF we must drive page table entries

- A kernel typically does not allow it (in fact kernel mitigation of this attack simply relies on having "invalid" page table entries set to proper values that do not map cacheable data)

- But what about a guest kernel?

- It can attack physical memory of the underlying host

- So it can also attack memory of co-located guests/VMs

- It is as simple as hacking the guest level page tables, on which an attacker that drives the guest may have full control

Attacking the host physical memory
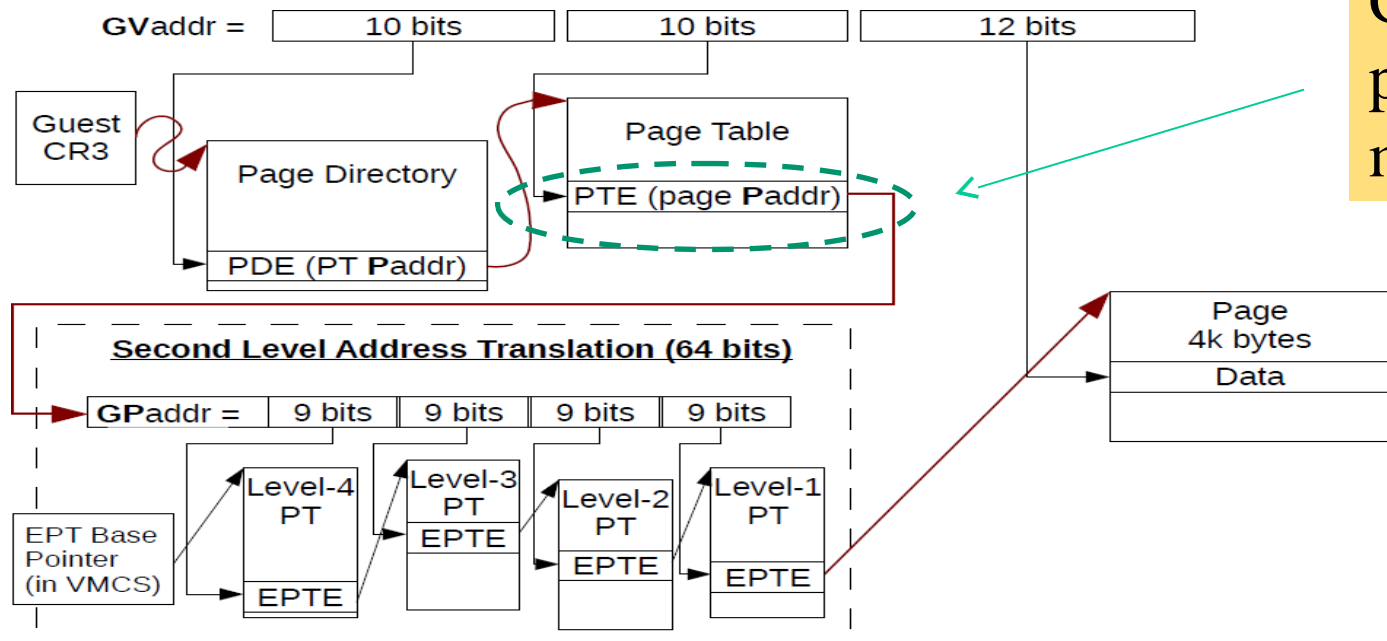
Change this to whatever physical address and make the entry invalid

Linux core map

- It is an array of `mem_map_t` (also known as `struct page`) structures defined in `include/linux/mm.h`

- The actual type definition is as follows (or similar along kernel advancement):

```
typedef struct page {

  struct list_head list;        /* ->mapping has some page lists. */

  ……

  atomic_t count;            /* Usage count, see below. */

  ……

  unsigned long flags;          /* atomic flags, some possibly
                        updated asynchronously */

  ……
} mem_map_t;
```

Linux free list data structures

- Free lists information is kept within the `pg_data_t` data structure defined in `include/linux/mmzone.h`, and whose actual instance is `contig_page_data`, which is declared in `mm/numa.c`

```
typedef struct pglist_data {
  struct zone node_zones[MAX_NR_ZONES];
  ……
  int nr_zones; //actually used zones
  ……
  struct page *node_mem_map;
  ……
} pg_data_t;
```
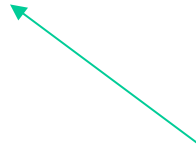
Describing a memory zone

Where we do pick free memory
blocks in a buddy allocator
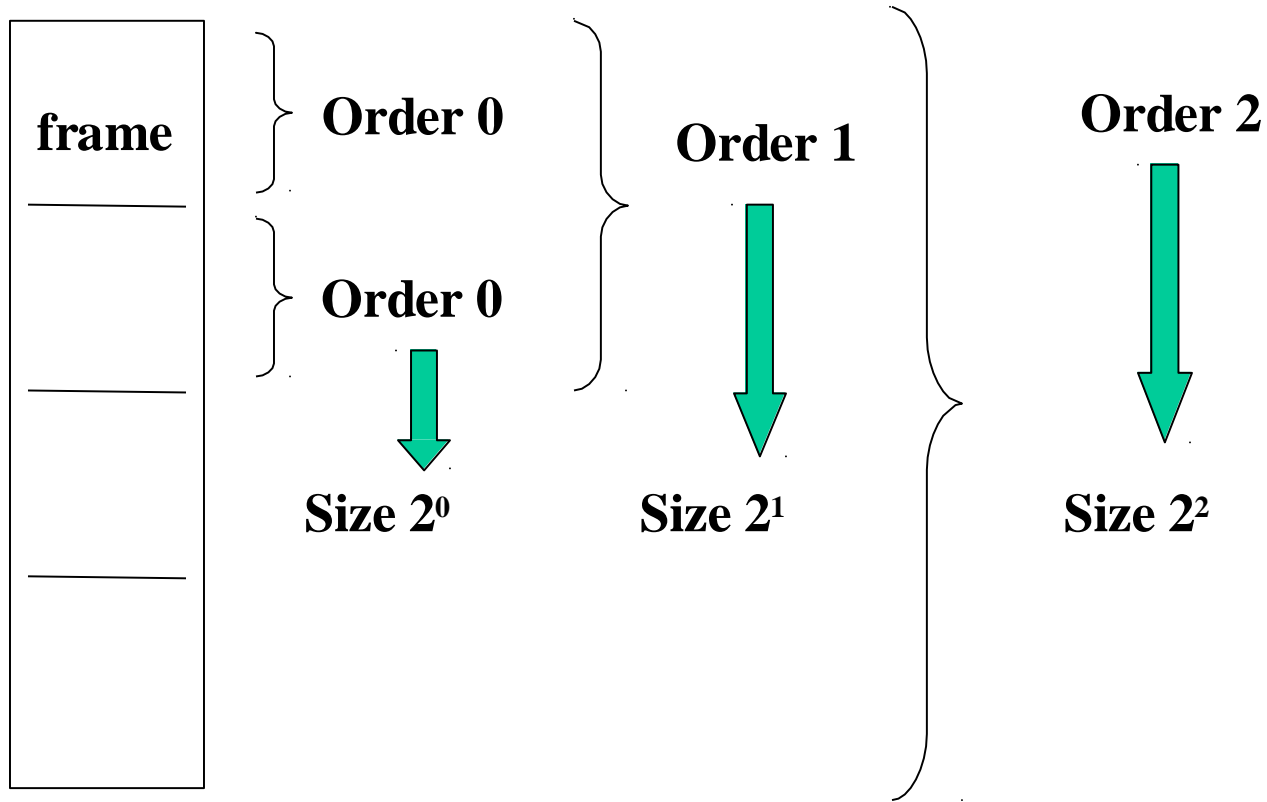
```
struct zone {
    ……

    free_area_t        free_area[MAX_ORDER];
    ……

    spinlock_t         lock;
    ……

    struct page        *zone_mem_map;

    ……

}
```
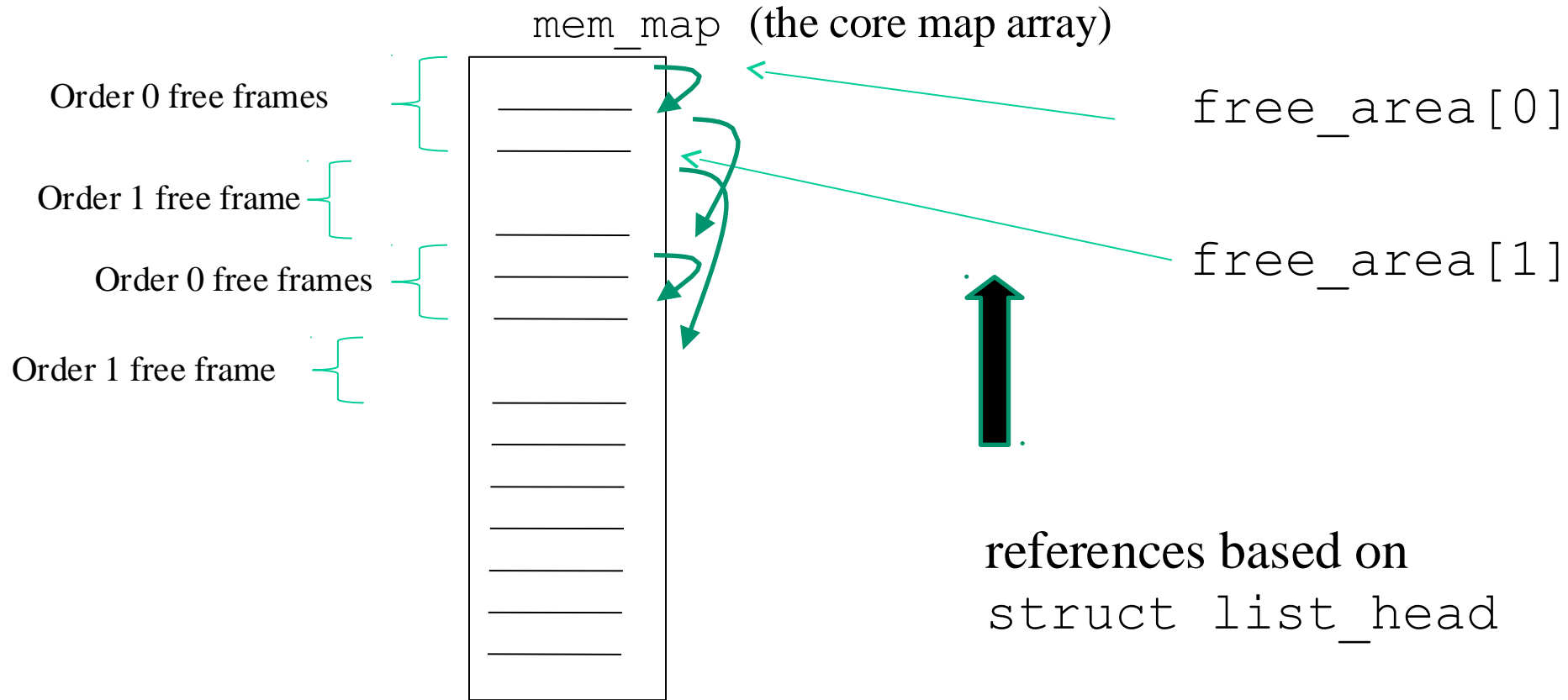
Up to 11 in recent
kernel versions
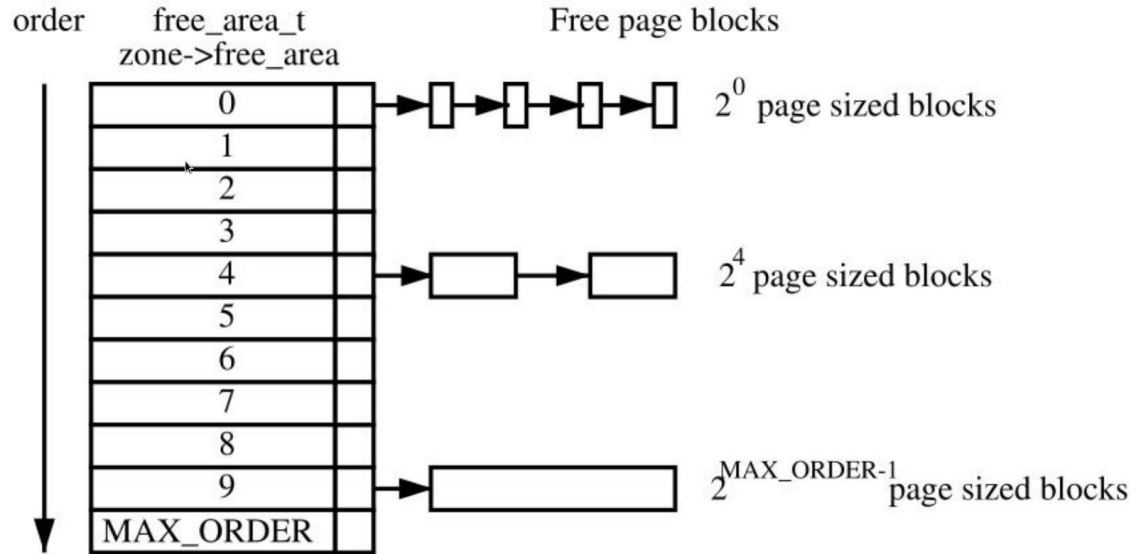(it was typically 5
before)

Buddy system features

frame

Order 0

Order 0

Order 1

Order 2

Size 2^0

Size 2^1

Size 2^2

Buddy allocation vs core map vs free list

mem_map (the core map array)

Order 0 free frames

Order 1 free frame

Order 0 free frames

Order 1 free frame

free_area[0]

free_area[1]

references based on
struct list_head

Recall that spinlocks are used to manage this data structure

A higher level view

order   free_area_t   Free page blocks
zone->free_area

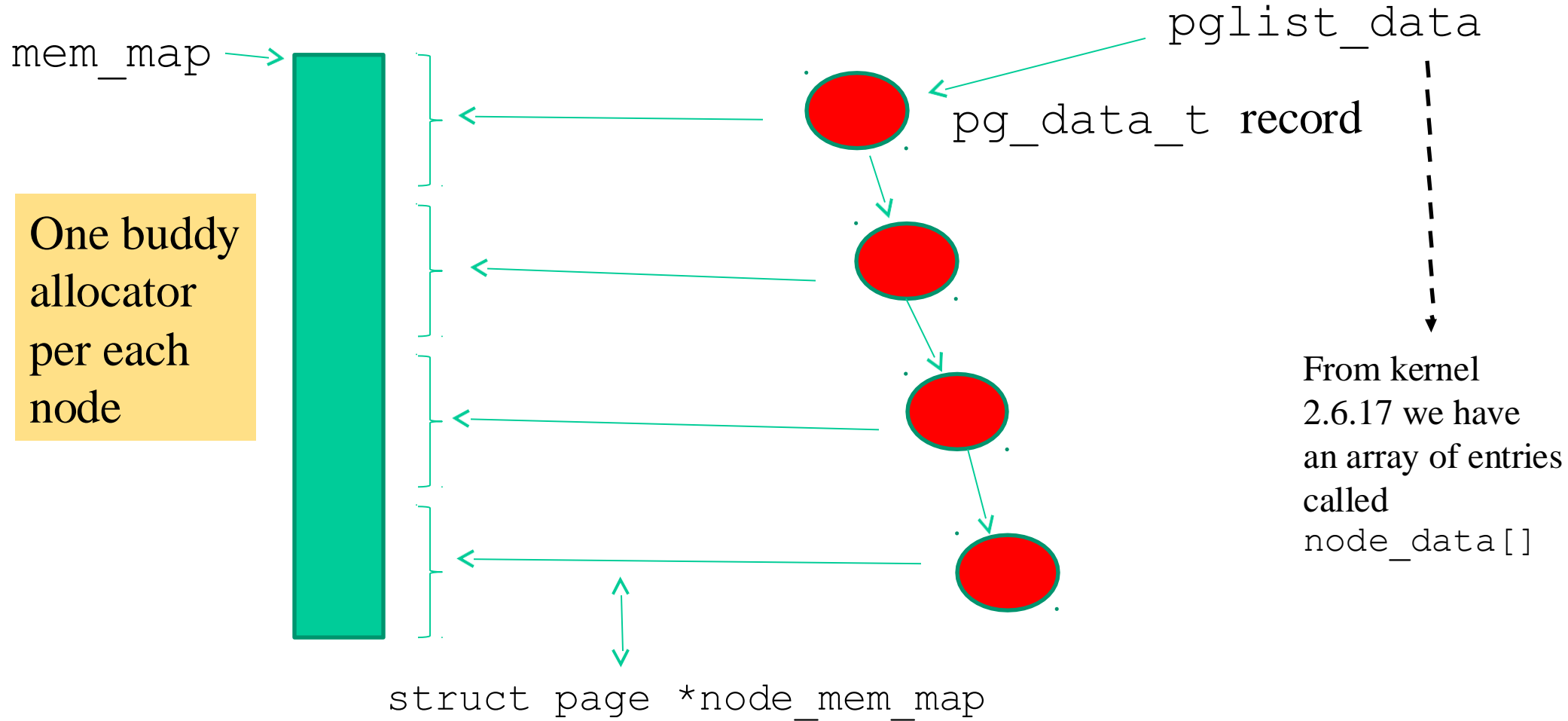| order		Free page blocks
0		2^0 page sized blocks
1		
2		
3		
4		2^4 page sized blocks
5		
6		
7		
8		
9		$2^{MAX\_ORDER-1}$ page sized blocks
MAX_ORDER		

A scheme (picture from: Understanding the Linux Virtual Memory Manager – Mel Gorman)
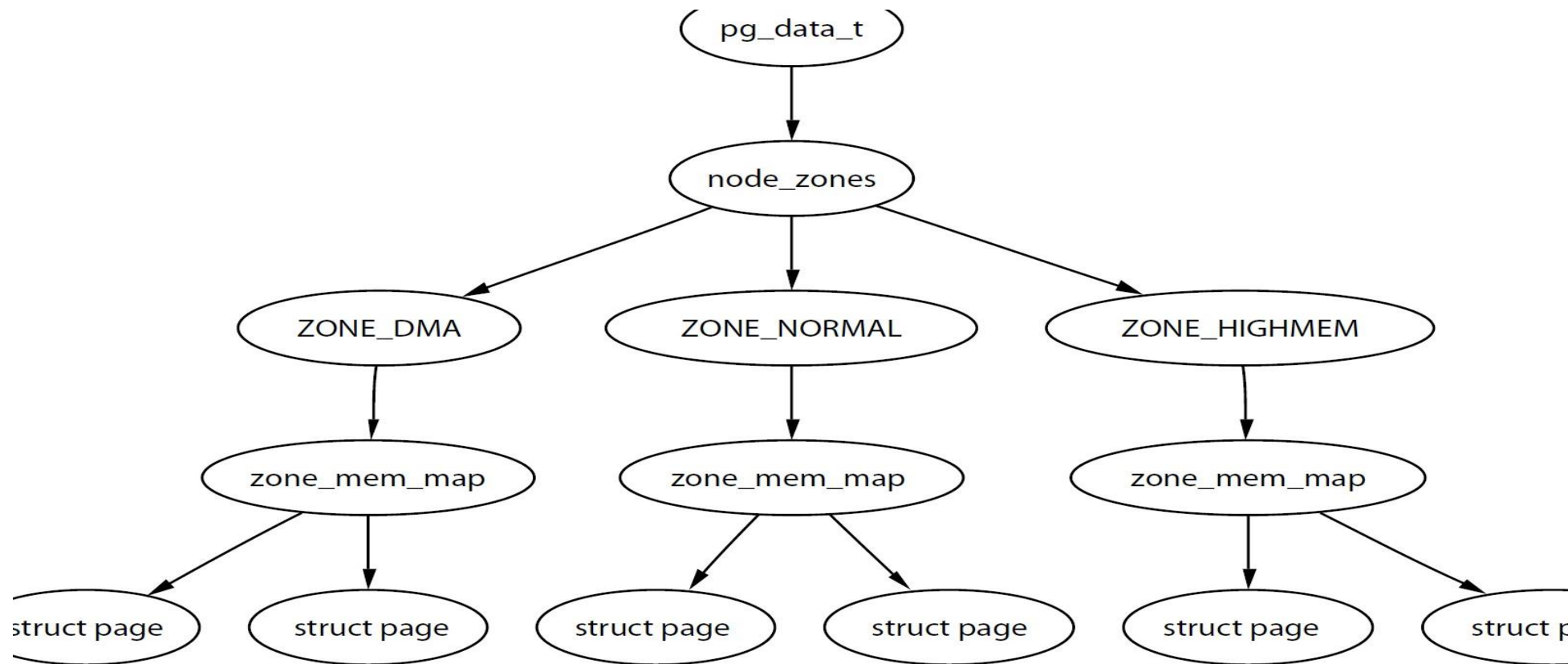
Jumping to NUMA aware Linux kernels (e.g. starting from kernel 2.6)

- The concept of multiple NUMA zones is represented by a `struct pglist_data` even if the architecture is Uniform Memory Access (UMA)

- This struct is always referenced by its `typedef pg_data_t`

- Every node in the system is kept on a NULL terminated list called `pgdata_list`, and each node is linked to the next with the field `pg_data_t→node_next`

- For UMA architectures like PC desktops, only one static `pg_data_t` structure is used

A scheme

mem_map →

```
One buddy
allocator
per each
node
```

pglist_data

pg_data_t record

From kernel
2.6.17 we have
an array of entries
called
`node_data[]`

`struct page *node_mem_map`

Allocation contexts - more generally, kernel level execution contexts

- Process context
  - Allocation is caused by a system call or a trap
    - Not satisfiable — wait is experienced along the current execution trace
    - Priority based schemes

- Interrupt
  - Allocation requested by an interrupt handler
    - Not satisfiable — no-wait is experienced along the current execution trace
    - Priority independent schemes

Buddy-system API

- After booting, the memory management system can be accessed via proper APIs, which drive operations on the aforementioned data structures

- The prototypes are in `#include <linux/malloc.h>`

- The very base allocation APIs are (bare minimal – page aligned allocation)

  - ➢ `unsigned long get_zeroed_page(int flags)`
    removes a frame from the free list, sets the content to zero and returns the virtual address

  - ➢ `unsigned long __get_free_page(int flags)`
    removes a frame from the free list and returns the virtual address
  - ➢ `unsigned long __get_free_pages(int flags, unsigned long order)`
    removes a block of contiguous frames with given `order` from the free list and returns the virtual address of the first frame

➢ `void free_page(unsigned long addr)`
  puts a frame into the free list again, having a given initial virtual address

➢ `void free_pages(unsigned long addr, unsigned long order)`
  puts a block of frames of given order into the free list again
  Note!!!!!!! Wrong order may give rise to kernel corruption in several kernel configurations

---

flags - used contexts

`GFP_ATOMIC` the call cannot lead to sleep (this is for interrupt contexts)

`GFP_USER` - `GFP_BUFFER` - `GFP_KERNEL` the call can lead to sleep

Buddy allocation vs direct mapping

- All the buddy-system API functions return <u>virtual addresses of directly mapped pages</u>

- We can therefore directly discover the position in memory of the corresponding frames

- Also, memory contiguity is guaranteed for both virtual and physical addresses

- Additonal flags could be used for memory allocation in specific zones (e.g. the ZONE_DMA flag)

Binding actual allocation to NUMA nodes

The real core of the Linux page allocator is the function

```
struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);
```

Hence the actual allocation chain is:

`__get_free_pages`

Node ID

mempolicy data

`alloc_pages_node`   (per NUMA node allocation)
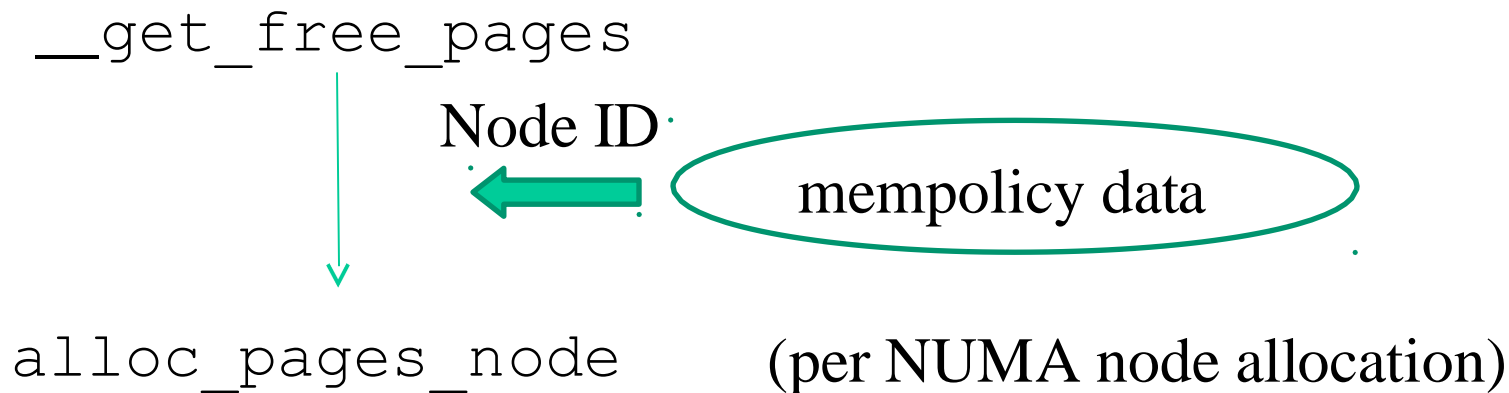
Mem-policy details

- Generally speaking, mem-policies determine what NUMA node needs to be involved in a specific allocation operation, which is thread specific

- Starting from kernel 2.6.18, the determination of mem-policies can be configured by the application code via system calls

Synopsis

```
#include <numaif.h>

int set_mempolicy(int mode, unsigned long *nodemask,
    unsigned long maxnode);
```

sets the NUMA memory policy of the calling process, which consists of a policy mode and zero or more nodes, to the values specified by the *mode*, *nodemask* and *maxnode* arguments. The *mode* argument must specify one of **MPOL_DEFAULT**, **MPOL_BIND**, **MPOL_INTERLEAVE** or **MPOL_PREFERRED**

… another example

Synopsis

```
#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.

… finally you can also move pages around

Synopsis
```
#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.

The case of frequent allocation/deallocation of target-specific data structures

- Here we are talking about allocation/deallocation operations of data structures
    1. that are used for a target-specific objective (e.g. in terms of data structures to be hosted)
    2. which are requested/released frequently

- The problem is that getting the actual buffers (pages) from the buddy system will lead to contention and consequent synchronization costs   (does not scale)

- In fact the (per NUMA node) buddy system operates with spinlock synchronized critical sections

- Kernel design copes with this issue by using pre-reserved buffers with lightweight allocation/release logic

… a classical example

- The allocation and deletion of page tables, at any level, is a very frequent operation, so it is important the operation is as quick as possible

- Hence the pages used for the page tables are cached in a number of different lists called *quicklists*

- For 3/4 levels paging, PGDs, PMDs/PUDs and PTEs have two sets of functions each for the allocation and freeing of page tables.

- The allocation functions are `pgd_alloc()`, `pmd_alloc()`, `pud_alloc()` and `pte_alloc()`, respectively the free functions are, predictably enough, called `pgd_free()`, `pmd_free,` `pud_free()` and `pte_free()`

- Broadly speaking, these APIs implement caching

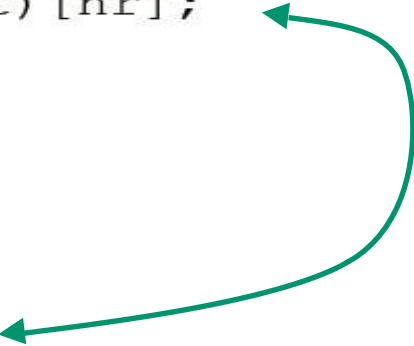# Actual quicklists (removed from kernel 5.12)

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

- They are implemented as a list of per-processing-unit page lists

- There is no need for synchronization

- If allocation fails, they revert to

$$\texttt{\_\_get\_free\_page()}$$

- In very latest versions of the Linux kernel pre-reserving is also done at the buddy allocator API

Quicklist API and algorithm

```c
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;
}
```

Beware these!!

Very recent quicklist vs buddy-system layering

- In more recent kernel versions the quicklist is one if the components of the buddy system

- It is used internally to this system, in particular for handling 4KB size allocation operations

- This allows using quicklists transparently by relying on the same API used for the buddy

- Quicklists operating at other levels are still available for kernel programming

Overall recent API – page grain allocation

- alloc_pages(gfp_mask, order)
- alloc_pages_node(node, gfp_mask, order)
- alloc_pages_exact(size, gfp_mask)

- __free_pages(page, order)
- free_pages(addr, order)
- free_pages_exact(addr, size)

 This uses caching and is backed on buddy-allocation

Overall recent API – page-fragment allocation

- page_frag_alloc(gfp_mask, size)
- page_frag_free(addr)

It is generally/largely used in networking stuff

It is not a general allocator, it is specific for less than page size buffers and typically relies on single page caching in the allocation scheme …. the SLAB/SLUB does not, let's check with it ...

SLAB (or SLUB) allocator - a cache of 'small' size buffers

- The main APIs are
  - ➢ `void *kmalloc(size_t size, int flags)`
    allocation of contiguous memory **of a given size** - it returns the virtual address

  - ➢ `void kfree(void *obj)`
    frees memory allocated via `kmalloc()`

  `kzalloc()` for zeroed buffers

- Main features
  - ➢ Cache aligned delivery of memory chunks (performance optimal access of data within the same chunk)
  - ➢ <u>Multiple caches associated with different allocation sizes</u>
  - ➢ Fast allocation/deallocation support

- Clearly, we can also perform node-specific requests via
  - ✓ `void *kmalloc_node(size_t size, int flags, int node)`

Details on cached allocation

- Slab is based on <u>pre-allocating pages</u> (a folio) from the buddy system

- The areas in these pages are then managed in a transparent manner by the slab system

- Cached-allocator instances for chunks of a given size can be created dynamically – this allows separation of memory usage for different services at kernel level

- The creation of a new cached allocator is only "virtual" (the real allocations can take place from another "equivalent allocator" (working with chunks of the same size)

- Any new cached allocator with "memory initialization" is never fused to existing ones

- A cached-allocator can be released when all its managed chunks have already been released

- You can check the existing allocators using */proc/slabinfo*

Cached allocation low level API - baseline

```
struct kmem_cache *kmem_cache_create(char *name,
                                     size_t size,
                                     size_t align,
                                     unsigned long flags,
                                      void (*ctl)(void *))
```

This is the memory initialization function

```
int kmem_cache_destroy(struct kmem_cache *cache)

void *kmem_cache_alloc(struct kmem_cache_t *cache, int prio)

void kmem_cache_free(struct kmem_cache_t *cache, void *ptr)
```

Actual management of the mem-cache

- When a mem-cache is created actual memory for allocation is not pre-reserved

- When a specific CPU tries to allocate from a given mem-cache, then memory is pre-reserved for that CPU

- Hence we have a per-CPU cache

- Therefore, a mem-cache has as many caches as the number of CPUs that attempted to allocate from that mem-cache

- The <u>free-list</u> of a mem-cache instance is a per-CPU list

- The <u>release-list</u> is accessed concurrently by the releasing CPUs (in non-blocking manner via CAS)

SLAB coloring

- A slab allocator is also assigned a color → this is a numerical code

- It is used to determine the position of the "first chunk" to be delivered into the slab

- Recall that the slab is a set of contiguous memory pages

- This allows mapping the first object of two different slabs for a same size on different cache lines with some (hopefully non-minimal) probability

- Clearly, the set of different colors is limited

Coloring details

- Suppose DSIZE is the dsize of metadata for a cached allocator

- Suppose it delivers chunks aligned to ALN

- Then assigning the color COL means that the first chunk of the slab is at the following offset from the beginning of the cached allocator → DSIZE + ALN * COL

- Essentially slab coloring means that the initial slab free areas are moved more or less close to the end of the used cached allocator areas

What about (very) large size allocations

- Classically employed while adding large size data structures to the kernel in a stable way

- We can go beyond the size-limit of the specific buddy system implementation

- This is the case when, e.g., mounting external modules

- **This time we are not guaranteed to get directly mapped pages**

- The main APIs are:
  - ➤ `void * vmalloc(unsigned long size);`

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

  - ➤ `void vfree(void * addr)`

    frees the above mentioned memory

kmalloc vs vmalloc - an overall reference picture

- Allocation size:
  - ➢ 128 KB for `kmalloc` (cache aligned)
  - ➢ 64/128 MB for `vmalloc`

- Physical contiguousness
  - ➢ Yes for `kmalloc`
  - ➢ No for `vmalloc`

- Effects on TLB
  - ➢ None for `kmalloc`
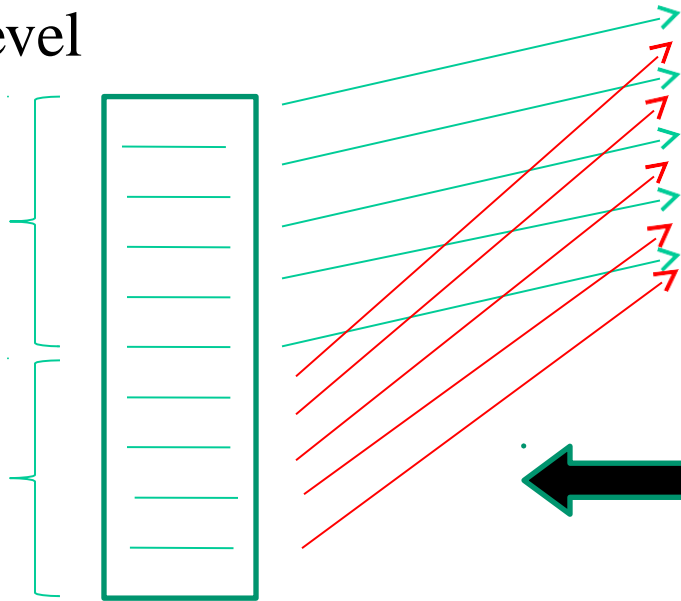 - ➢ Global for `vmalloc` (transparent to `vmalloc` users)

vmalloc operations (i)

- Based in remapping a range of contiguous pages in (non contiguous) physical memory

Busy frame
Free frame

Kernel level pages
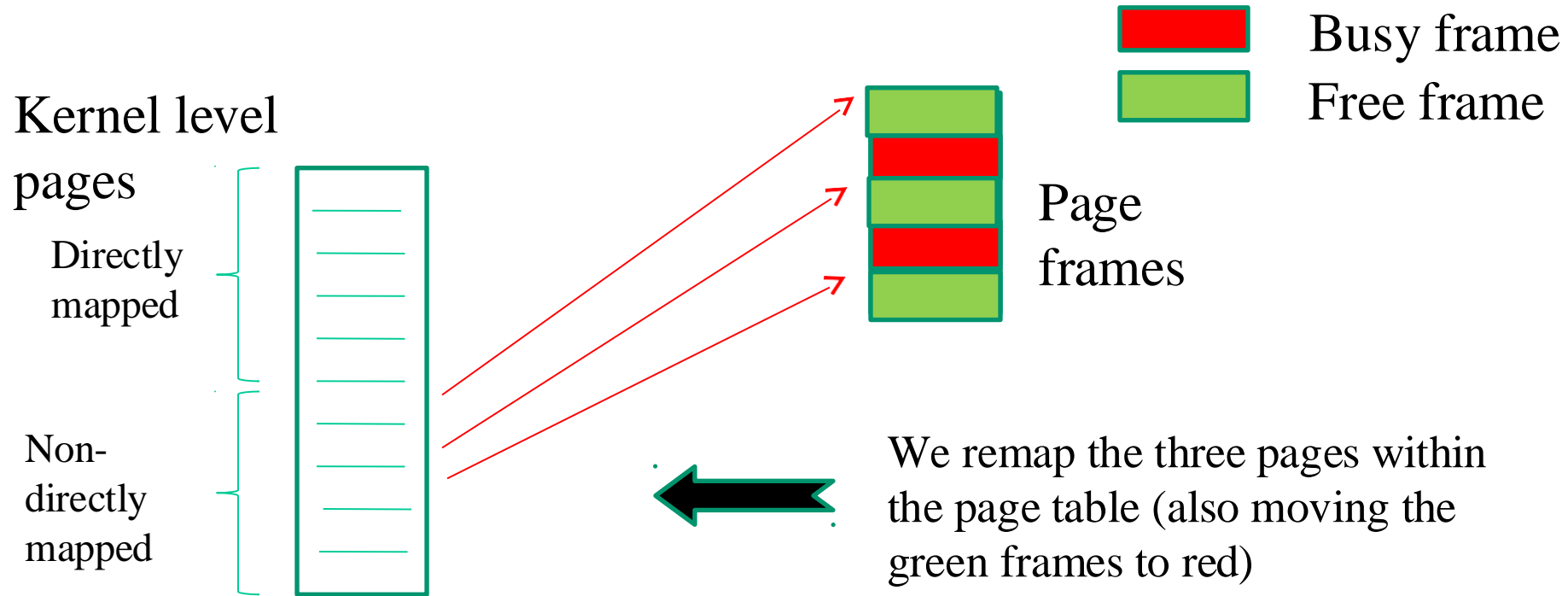
Directly mapped

Non-directly mapped

Page frames

Suppose we need 3 contiguous virtual pages

vmalloc operations (ii)

Busy frame

Free frame

Kernel level pages

Directly mapped

Non-directly mapped

Page frames

We remap the three pages within the page table (also moving the green frames to red)

Clearly with `vmalloc` we typically remap much larger blocks of pages

Kernel-page remapping vs hardware state

- Kernel-page mapping has a "global nature"

- Any core can use the same mapping, supported by the same page tables

- When running `vmalloc/vfree` services on a specific core, <u>all the other cores need to observe the updated mapping</u>

- Cached mappings within TLBs need therefore to be updated via proper operations

TLB implicit vs explicit operations

- The level of automation in the management process of TLB entries depends on the specific hardware architecture

- Kernel hooks have to exist for explicit management of TLB operations (these are compile-time mapped to null operations in case of fully automated TLB management)

- For x86 processors automation is only partial

- Specifically, 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 within 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 global TLB flush

```
void flush_tlb_all(void)
```

- This flushes the entire TLB **<u>on all processors running in the system</u>**, which makes it the most expensive TLB flush operation

- After it completes, all modifications to the page tables will be visible globally

- This is required after the kernel page tables, which are global in nature, have been modified

- Examples are `vmalloc()`/`vfree()` operations

Linux global TLB flush vs x86

- x86 does not offer pure hardware support for flushing all the TLBs on board of the architecture

- It offers a baseline mechanism to let CPU-cores coordinate

- A software layer is used to drive what to do while coordinating (namely TLB invalidation)

- We will come back to this issue when analyzing actual interrupt achitectures on multi-core machines

The x86 timeline of vmalloc

Invocation (on some generic CPU-core)

- Acquire memory from the buddy allocator
- Update kernel page table

return

Cross CPU-core coordination for TLB invalidation (via CR3 rewriting)
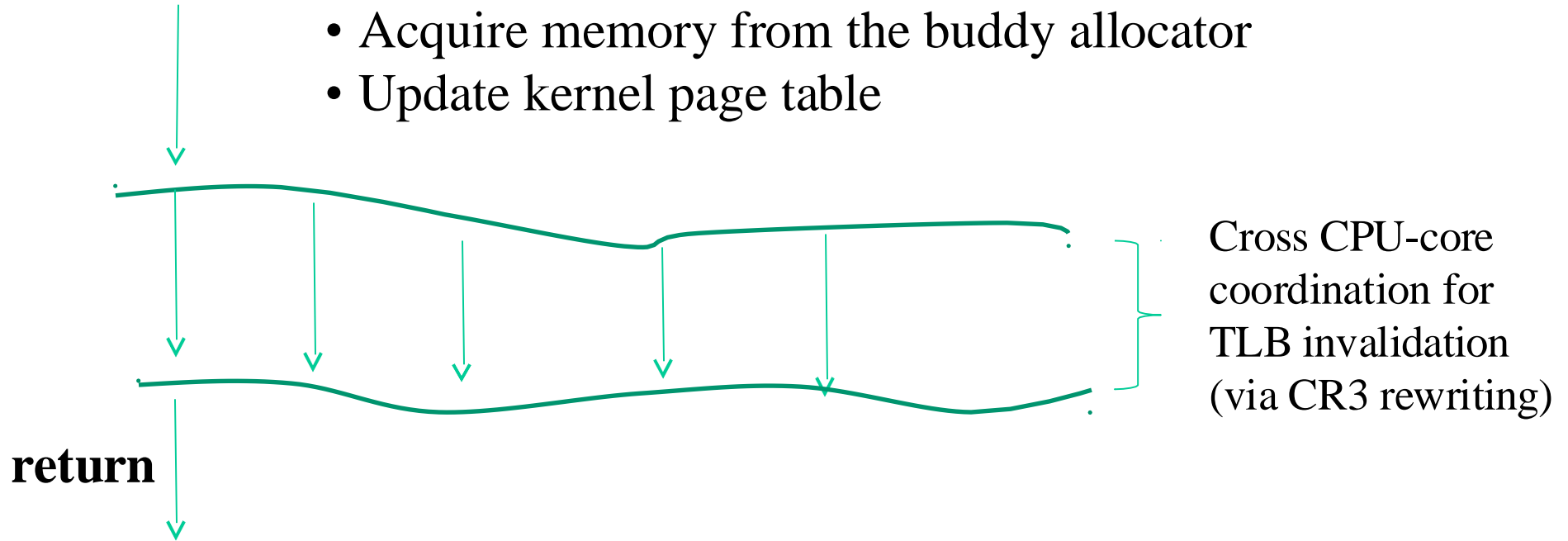
Linux partial TLB flush

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

- This flushes all TLB entries related to the userspace portion for the requested mm context

- This is only called when an operation has been performed that affects the entire address space

- e.g., after all the address mapping has been duplicated with `dup_mmap()` for fork or after all memory mappings have been deleted with `exit_mmap()`

- Interaction with COW protection

```
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 (e.g. `mremap()`) or when changing permissions (e.g. `mprotect()`)

- This API is provided for architectures that can remove ranges of TLB entries quickly rather than iterating with `flush_tlb_page()`

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

- This API is responsible for flushing a single page from the TLB

- The two most common uses of it are for flushing the TLB after a page has been faulted in or has been paged out
    - ✓ Interactions with page table access firmware

x86 partial TLB invalidation

INVLPG

Invalidate TLB Entry

Opcode	Mnemonic	Description
0F 01/7	INVLPG m	Invalidate TLB Entry for page that contains m.

Description

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVLPG instruction normally flushes the TLB entry only for the specified page; however, in some cases, it flushes the entire TLB. See "MOV-Move to/from Control Registers" in this chapter for further information on operations that flush the TLB.

Operation

```
Flush(RelevantTLBEntries);
ContinueExecution();
```

Flags affected

None.

IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

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

- ➤ This API is called when the page tables are being torn down and freed

- ➤ Some platforms cache the lowest level of the page table, i.e., the actual page frame storing entries, 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

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

➢ This API is 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

➢ In some case it is used for **preloading TLB entries (e.g. like in ARM Cortex processors)**