Advanced Operating Systems
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. Boot and steady state behavior of the memory management system in the Linux kernel

Basic terminology

- **firmware**: a program coded on a ROM device, which can be executed when powering a processor on
- **bootsector**: predefined device (e.g. disk) sector keeping executable code for system startup
- **bootloader**: the actual executable code loaded and launched right before giving control to the target operating system
  - this code is partially kept within the bootsector, and partially kept into other sectors
  - It can be used to parameterize the actual operating system boot

Startup tasks

- The firmware gets executed, which loads in memory and launches the bootsector content
- The loaded bootsector code gets launched, which may load other bootloader portions
- The bootloader ultimately loads the actual operating system kernel and gives it control
- The kernel performs its own startup actions, which may entail architecture setup, data structures and software setup, and process activations
- To emulate a steady state unique scenario, at least one process is derived from the boot thread (namely the IDLE PROCESS)

Where is memory management there?

- All the previously listed 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

Traditional firmware on x86

- It is called BIOS (Basic I/O System)
- Interactive mode can be activated via proper interrupts (e.g. the F1 key)
- Interactive mode can be used to parameterize firmware execution (the parameterization is typically kept via CMOS rewritable memory devices powered by apposite temporary power suppliers)
- The BIOS parameterization can determine the order for searching the boot sector on different devices
- A device boot sector will be searched for only if the device is registered in the BIOS list

Bios bootsector

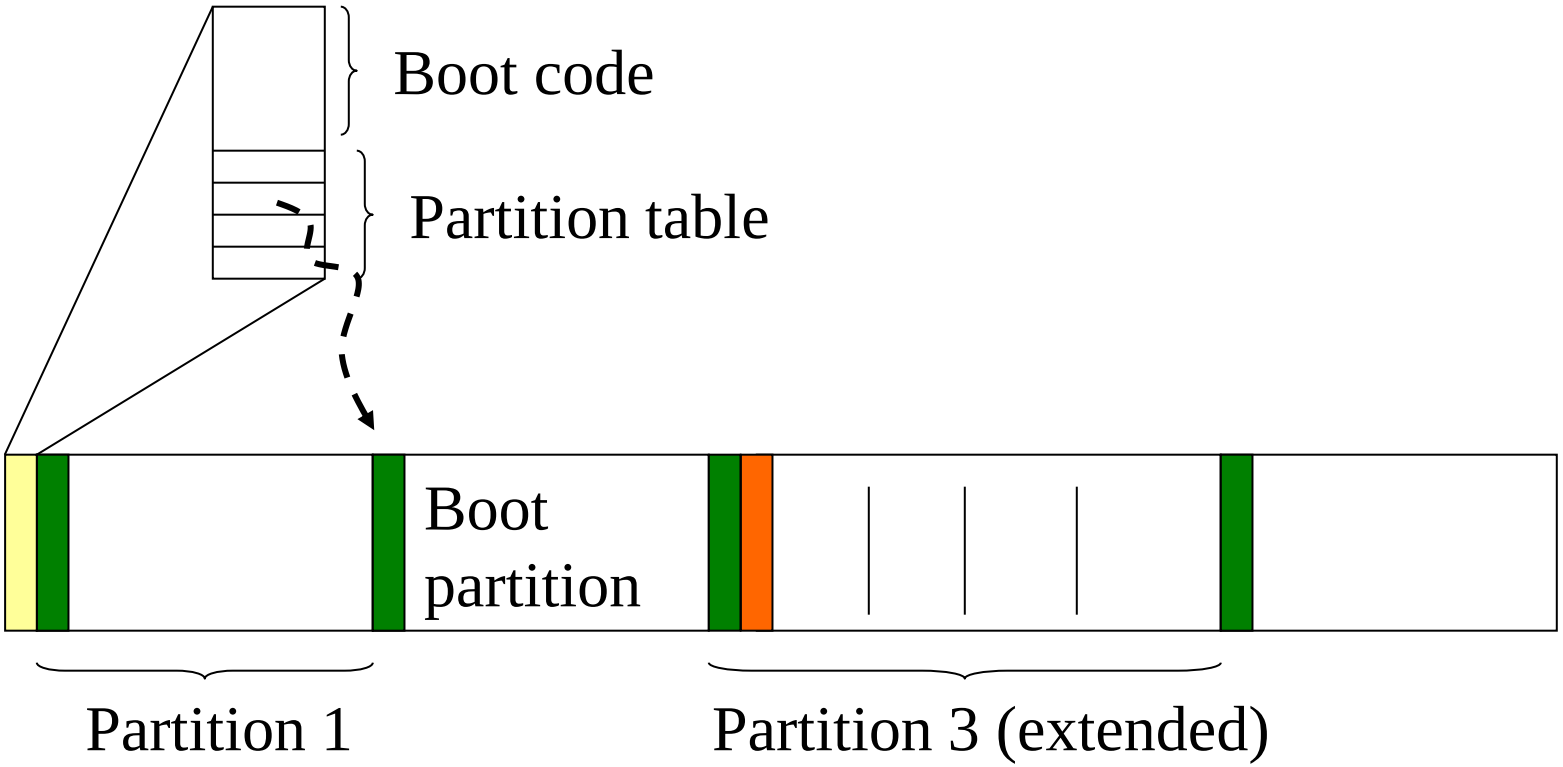
- The first device sector keeps the so called **master boot record** (MBR)
- This sector keeps executable code and a 4-entry tables, each one identifying a different device partition (in terms of its positioning on the device)
- The first sector in each partition can operate as the partition boot sector (BS)
- In case the partition is extended, then it can additionally keep up to 4 sub-partitions (hence the partition boot sector can be structured to keep an additional partitioning table)
- Each sub-partition can keep its own boot sector

RAM image of the MBR

Offset	Size (bytes)	Description
0	436 (to 446, if you need a little extra)	MBR **Bootstrap** (flat binary executable code)
0x1b4	10	Optional "unique" disk ID[1]
0x1be	64	MBR **Partition Table**, with 4 entries (below)
0x1be	16	First partition table entry
0x1ce	16	Second partition table entry
0x1de	16	Third partition table entry
0x1ee	16	Fourth partition table entry
0x1fe	2	(0x55, 0xAA) "Valid bootsector" signature bytes

Grub (if you use it) or others

An example scheme with Bios



Boot sector

Extended partition boot sector

Nowadays huge limitation: the maximum size of manageable disks is 2TB

UEFI – Unified Extended Firmware Interface

- It is the new standard for basic system support (e.g. boot management)
- It removes several limitations of BIOS:
  - We can (theoretically) handle disks up to 9 zettabytes
  - It has a more advanced visual interface
  - It is able to run EFI executables, rather than simply loading and launching the MBR code
  - It offers interfaces to the OS for being configured (rather than being exclusively configurable by triggering its interface with Fn keys at machine startup)

UEFI device partitioning

- Based on GPT (GUID Partition Table)
- GUID = Globally Unique Identifier ….. Theoretically all over the world (if the GPT has in its turn a unique identifier)
- Theoretically unbounded number of partitions kept in this table – No longer we need extended partitions for enlarging the partitions' set
- GPT are replicated so that if a copy is corrupted then another one will work – this breaks the single point of failure represented by MBR and its partition table

Bios/UEFI tasks upon booting the OS kernel (i)

- The bootloader/EFI-loader, e.g., GRUB, loads in memory the initial image of the operating system kernel
- This includes a ``machine setup code'' that needs to run before the actual kernel code takes control
- This happens since a kernel configuration needs given setup in the hardware upon being launched
- The machine setup code ultimately passes control to the initial kernel image

Bios/UEFI tasks upon booting the OS kernel (ii)

- In Linux, this kernel image executes starting from the `start_kernel()` in `init/main.c`
- This kernel image is way different, both in size and structure, from the one that will operate at steady state
- Just to name one reason, boot is typically highly configurable!

What about Linux boot on multi-core/HT machines

- The `start_kernel()` function is executed along a single CPU-core (the master)
- All the other cores (the slaves) only keep waiting that the master has finished
- The kernel internal function `smp_processor_id()` can be used for retrieving the ID of the current core
- This function is based on ASM instructions implementing a hardware specific ID detection protocol
- This function operates correctly either at kernel boot or at steady state

The actual support for CPU-core identification

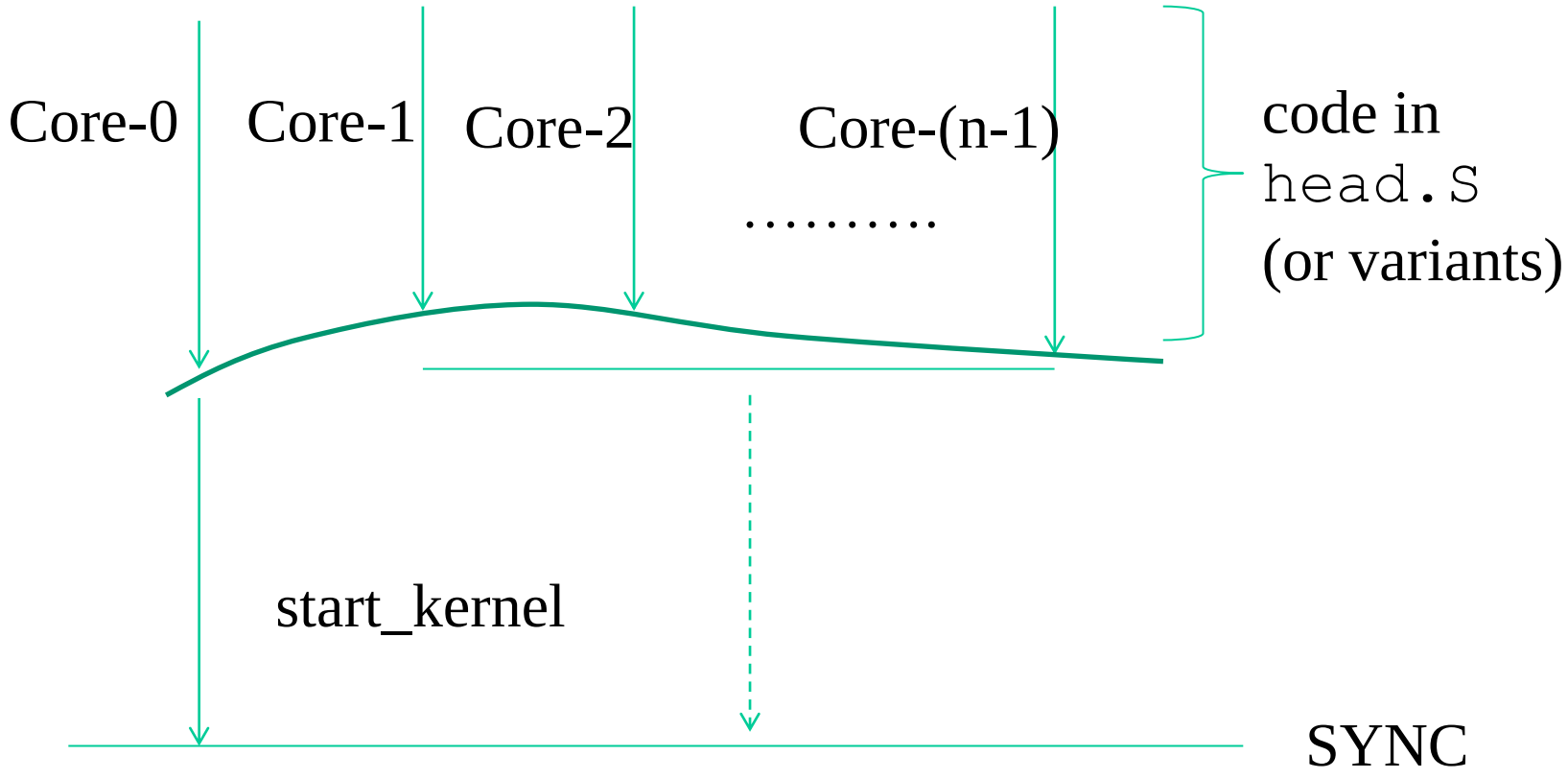
x86 Instruction Set Reference

CPUID

CPU Identification

Opcode	Mnemonic	Description
0F A2	CPUID	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, according to the input value entered initially in the EAX register.

Actual kernel startup scheme in Linux



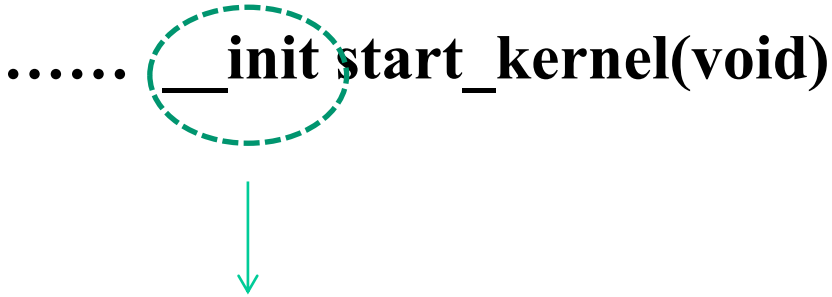
An example `head.S` code snippet: triggering Linux paging (IA32 case)

```
/* * 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 / kernel 5)

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

……. /* here in the middle we account for other stuff like
            randomization */

movq %rax, %cr3
```

Hints on the signature of the `start_kernel` function (as well as others)



The reason for this is to recover main memory storage which is relevant because of both:

- Reduced available RAM (older configurations)
- Increasingly complex (and hence large in size) startup code

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

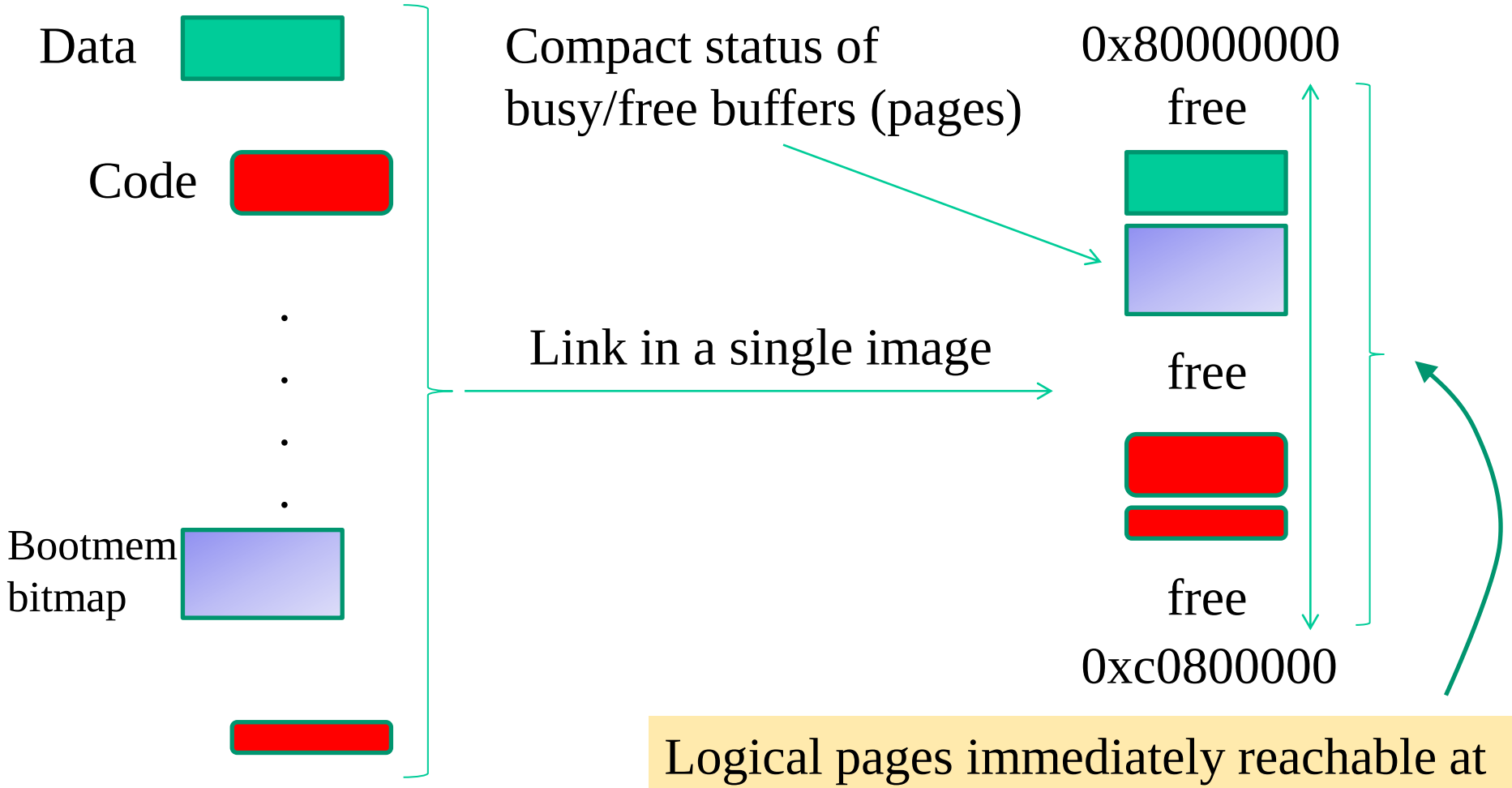
Management of `__init` functions

- The kernel 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

An exemplified picture of bootmem



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 an address falling into that page
- The point is that the expressed address is typically a virtual one (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)
- The initial kernel image has a page table, with a minimum number of pages mapped to RAM, those handled by the bootmem subsystem

How is RAM memory 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

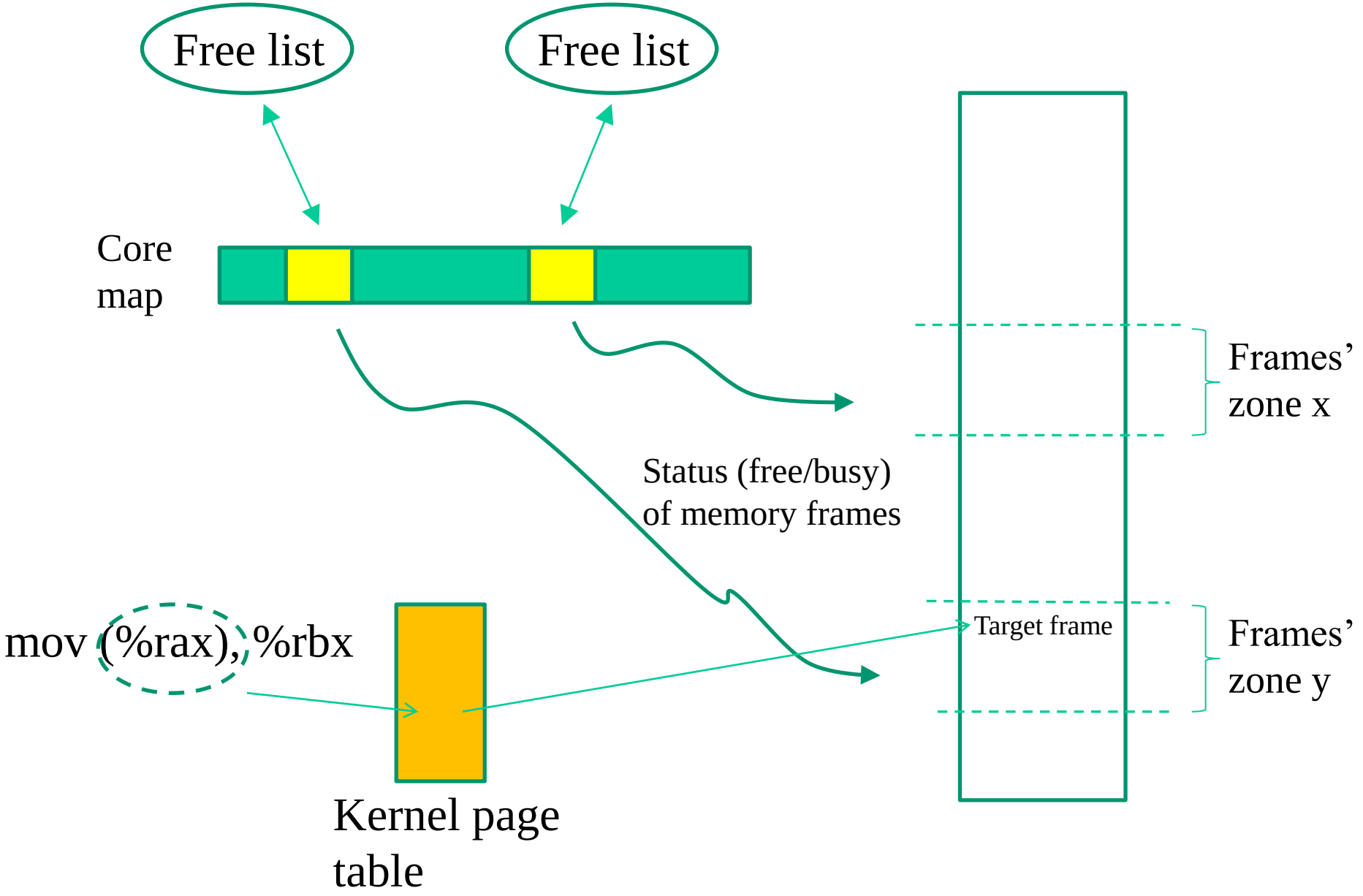
- In more recent versions of OS kernels the bootmem architecture has been enriched
- It allows for keeping track of free/busy frames with a per-NUMA node granularity
- The newer architecture is called “memblock” in Linux
- An additional logic is inserted for setting up the memblock system to indicate 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

Actual kernel data structures for managing memory

- Kernel 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

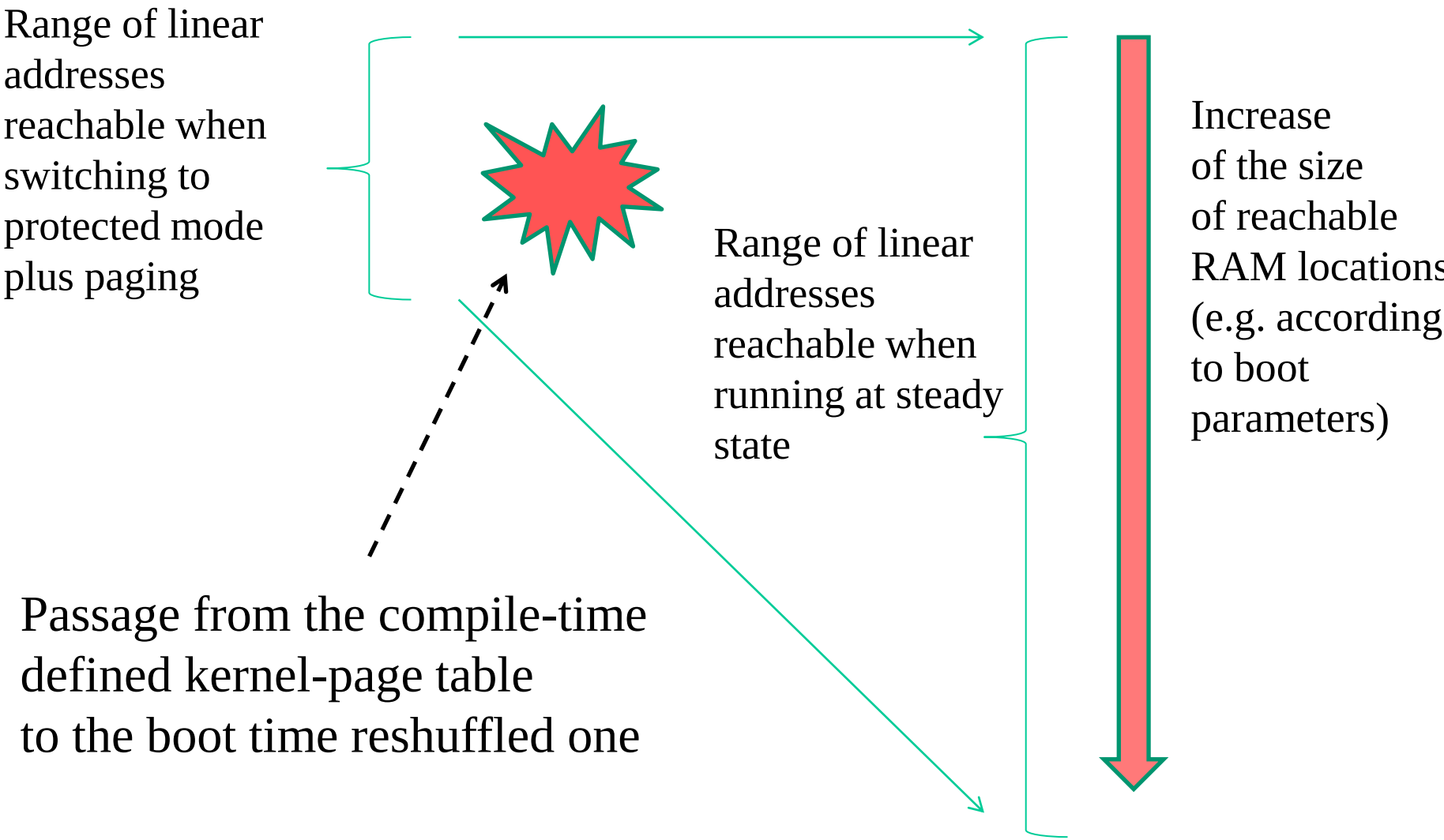
A scheme


Free list
Free list
Core map
Status (free/busy) of memory frames
Frames' zone x
Frames' zone y
Target frame
mov (%rax), %rbx
Kernel page table

Objectives of the kernel page table setup

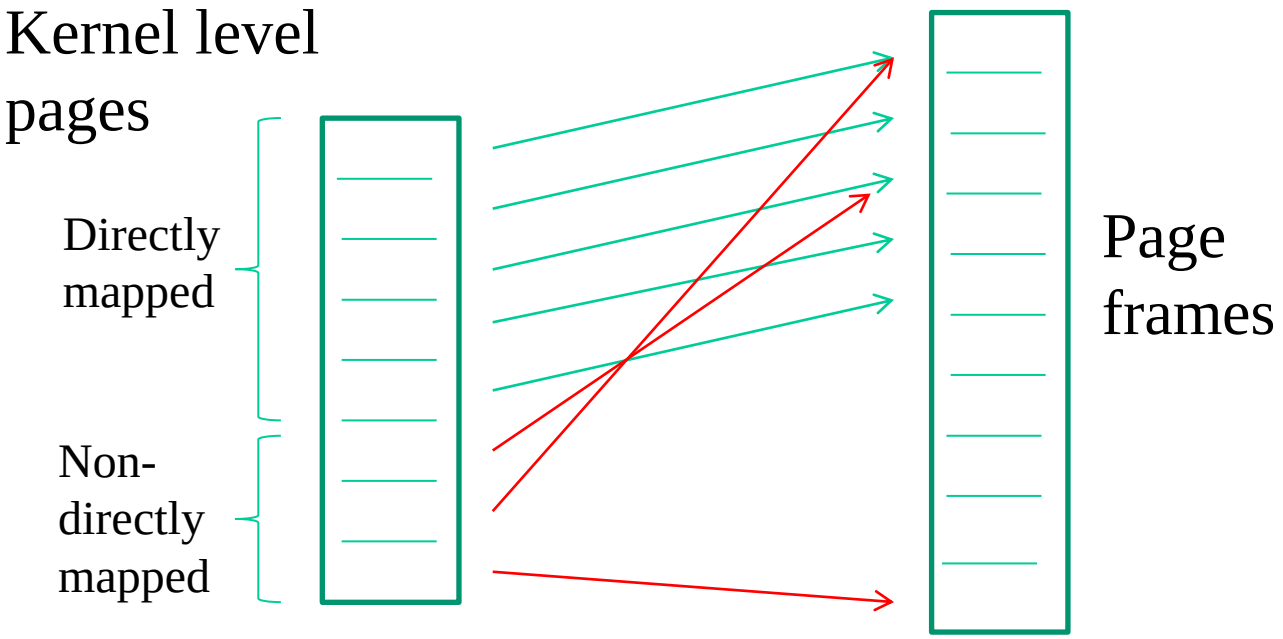
- These are basically two:
  - ✓ Allowing the kernel software to use virtual addresses while executing (either at startup or at steady state)
  - ✓ Allowing the kernel software (and consequently the application software) to reach (in read and/or write mode) the maximum admissible (for the specific machine) 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, e.g., given that the available RAM to drive can be parameterized

A scheme


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 = ψ(VA) where ψ is (typically) a simple function subtracting a predetermined constant value to VA
- Not all the kernel level virtual pages are directly mapped



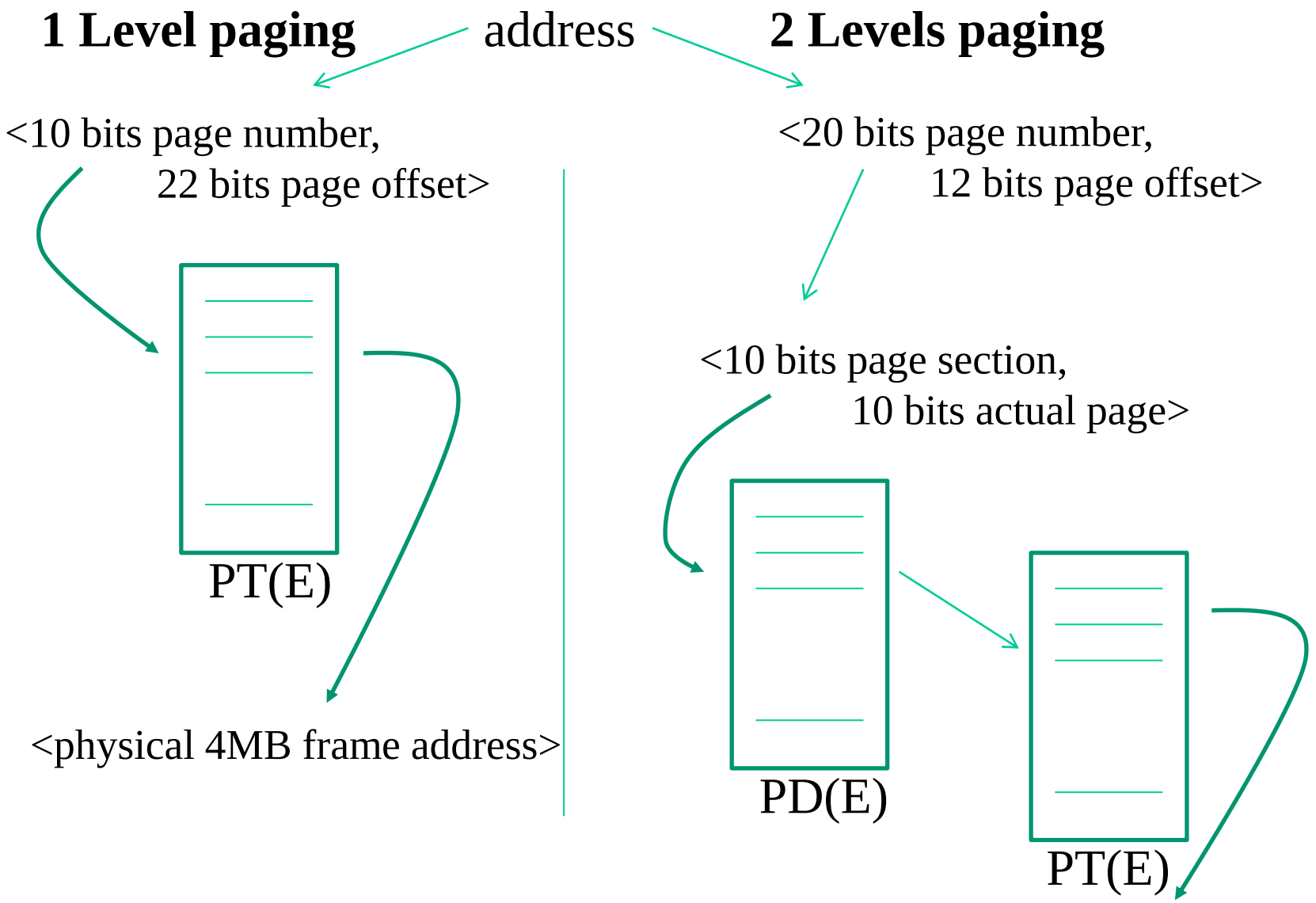
Virtual memory vs boot sequence: the i386 very didactical example

- 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 actual paging rule (namely the page granularity and the number of paging levels – up to 2 in i386) is identified via proper bits within the entries of the page table
- The physical address of the setup page table is kept within the CR3 register

Virtual memory vs boot sequence: the i386 very didactical example

- The steady state paging scheme used by LINUX will be activated during the kernel boot procedure
- The max size of the address space for LINUX processes on i386 machines (or protected mode) is 4 GB
  - 3 GB are within user level segments
  - 1 GB is within kernel level segments

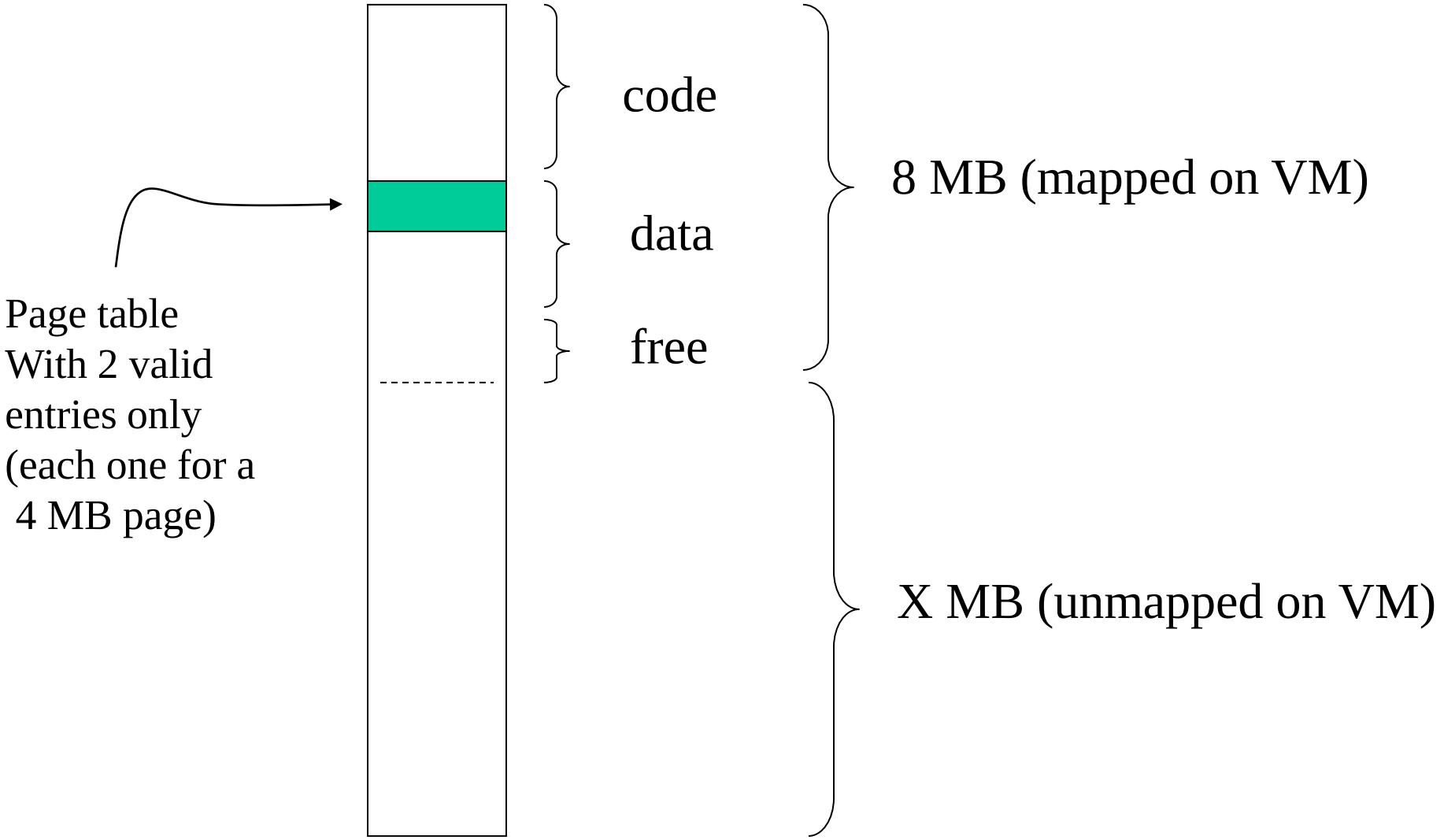
Details on the page table structure in i386 (i)



Details on the page table structure in i386 (ii)

- It is allocated in physical memory into 4KB blocks, which can be non-contiguous
- In typical LINUX configurations, once set-up it maps 4 GB addresses, of which 3 GB at null reference and (almost) 1 GB onto actual physical memory
- Such a mapped 1 GB corresponds to kernel level virtual addressing and allows the kernel to span over 1 GB of physical addresses
- To drive more physical memory, additional configuration mechanisms need to be activated, or more recent processors needs to be exploited as we shall see

i386 memory layout at kernel startup: still very didactical for kernel 2.4



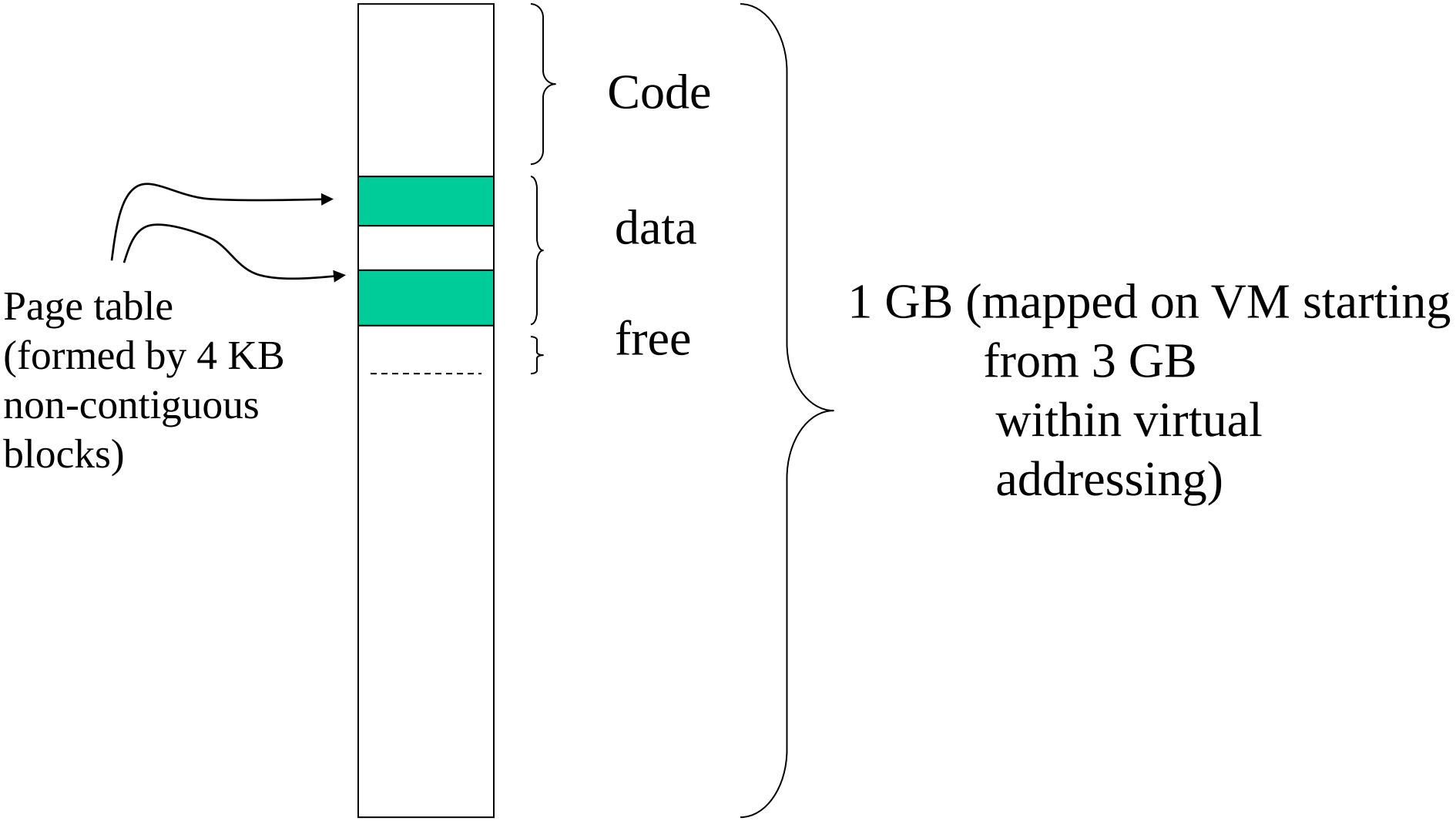
Actual issues to be tackled

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)

Back to the concept of *bootmem*

1. Memory occupancy and location of the initial kernel image is determined by the compile/link process
2. As we have seen, a compile/link time memory manager is embedded into the kernel image, which is called bootmem manager
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 (to be employed at boot time) in order to get free buffers
5. These buffers are sets of contiguous (or single) page aligned areas
6. As hinted, this subsystem is in charge of handling `_init` marked functions in terms of final release of the corresponding buffers

Kernel page table collocation within physical memory

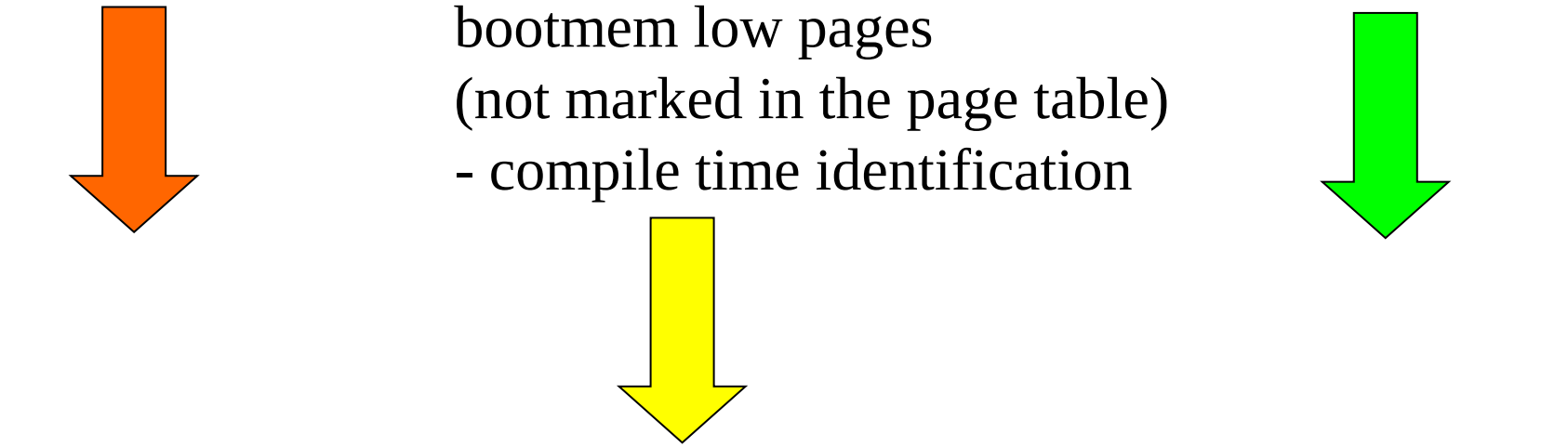


Low level “pages”

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

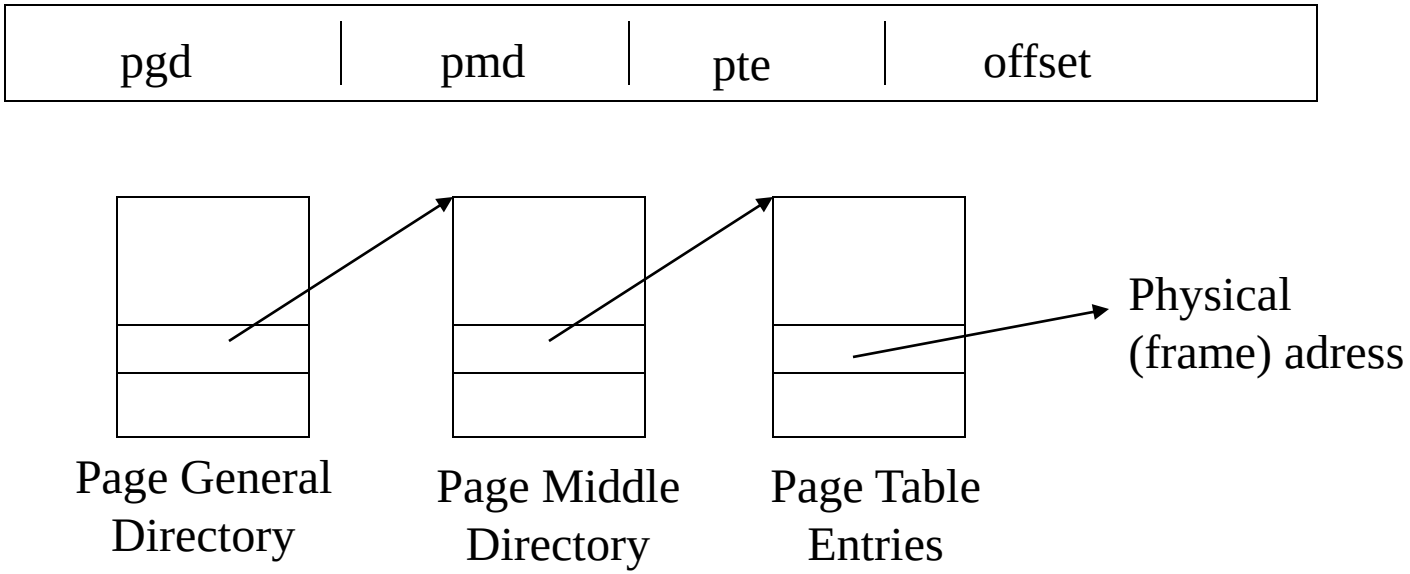
Expand page table via
bootmem low pages
(not marked in the page table)
- compile time identification

Finalize kernel handled
page size (4KB)



LINUX paging vs i386

- LINUX virtual addresses exhibit (at least) 3 indirection levels



- On i386 machines, paging is supported limitedly to 2 levels (`pde`, page directory entry – `pte`, page table entry)
- **Such a dicotomy is solved by setting null** the `pmd` field, which is proper of LINUX, and mapping
  - `pgd` LINUX to i386 `pde`
  - `pte` LINUX to i386 `pte`

i386 page table size

- Both levels entail 4 KB memory blocks
- Each block is an array of 4-byte entries
- Hence we can map **1 K x 1K pages**
- Since each page is 4 KB in sixe, we get a 4 GB virtual addressing space
- 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

Page table 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`
- They are

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

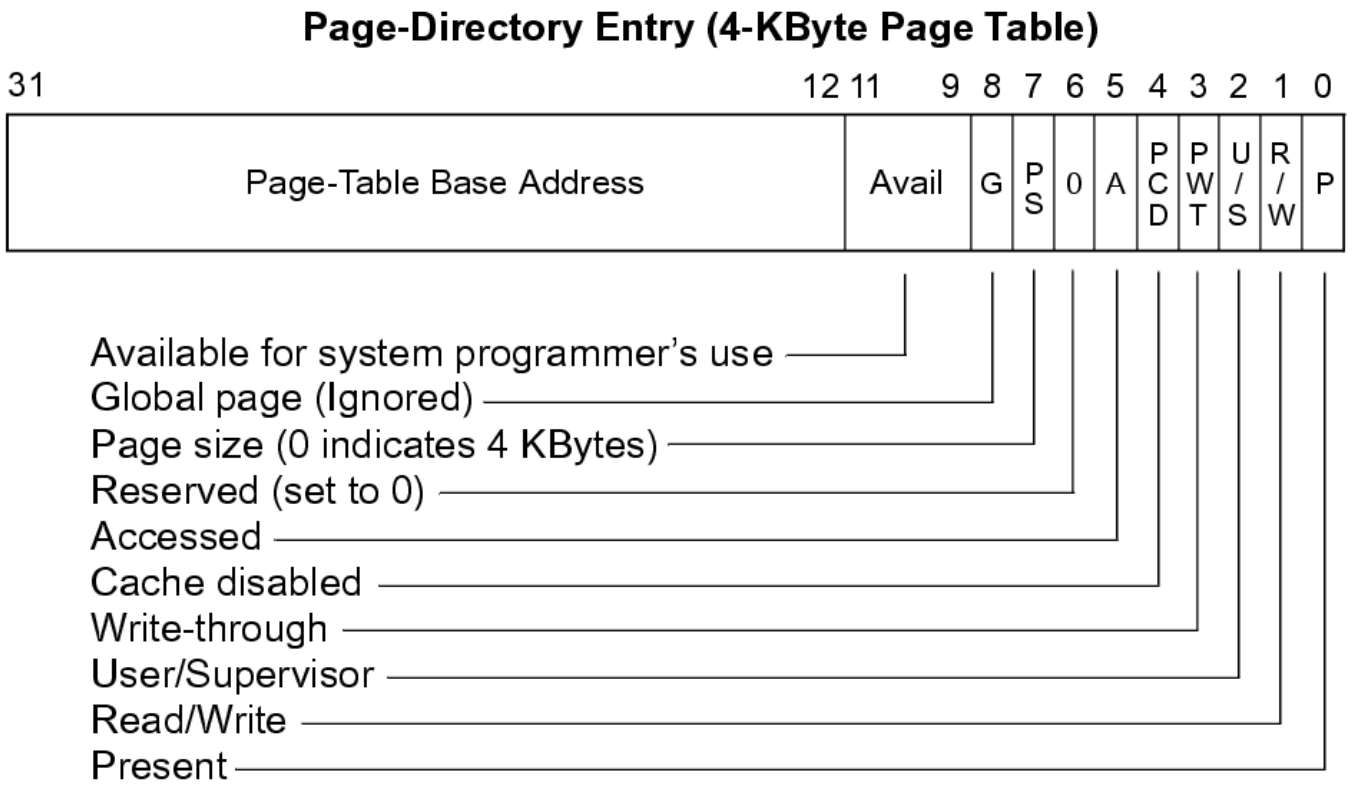
Debugging

- 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;
```

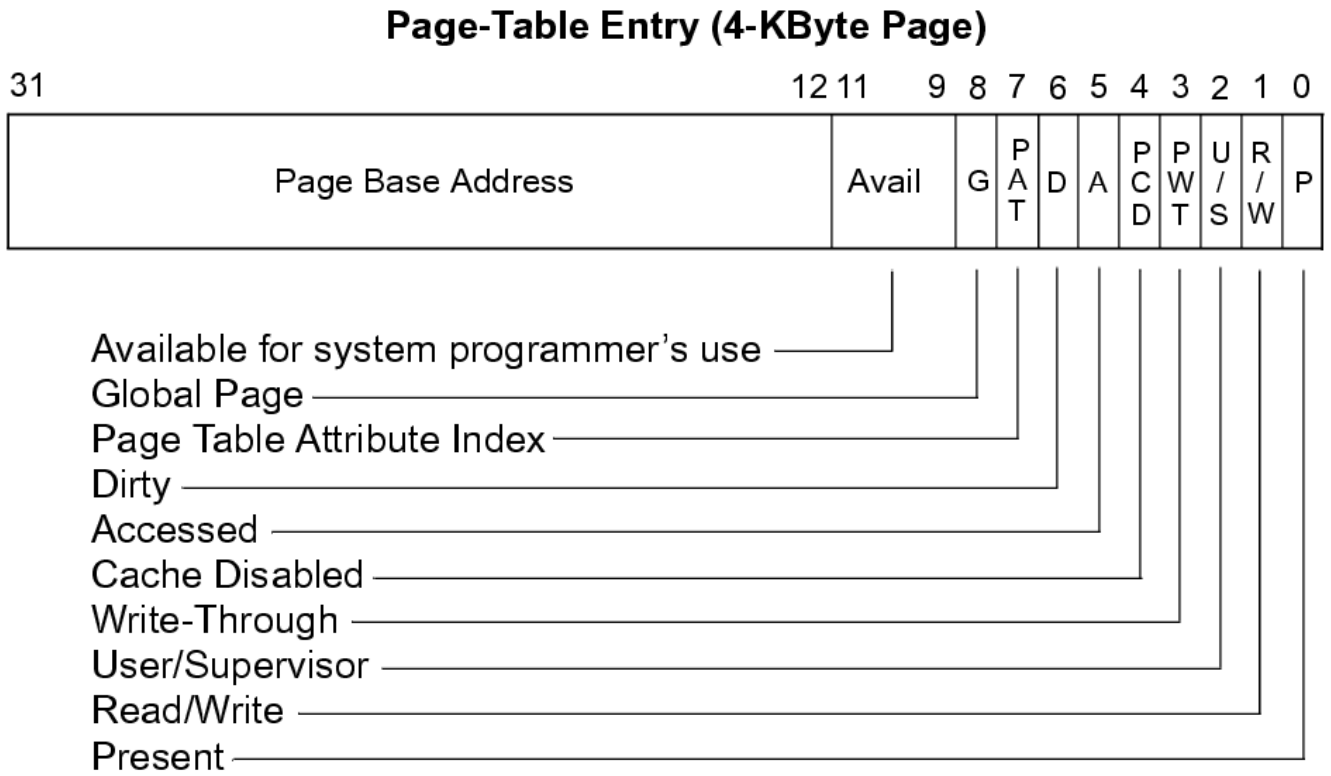
  the compiler enables assignments such as `x=y` and `y=x`
- Hence, there is the need for defining different structured types which simulate the base types that would otherwise give rise to compiler equivalent aliases

i386 PDE entries



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

i386 PTE entries



Nothing tells whether we can fetch (so 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**

Bit masking

- 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 */
} ;
```

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
- **Re-execution might give rise to additional traps, depending on firmware checks on the page table**
- As an example, the attempt to access a read only page in write mode will give rise to a trap (which triggers the **segmentation fault handler**)

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);
}
```

Kernel page table initialization details

- As said, the kernel PDE is accessible at the virtual address kept by `swapper_pg_dir` (now `init_level4_pgt` on x86-64/kernel3 or `init_top_pgt` on x86-64/kernel4-5)
- The room for PTE tables gets reserved within the 8MB of RAM that are accessible via the initial paging scheme
- Reserving takes place via the macro `alloc_bootmem_low_pages()` which is defined in `include/linux/bootmem.h` (this macro returns a virtual address)
- Particularly, it returns the pointer to a 4KB (or 4KB x N) buffer which is page aligned
- This function belongs to the (already hinted) basic memory management subsystem the boot lies on

Kernel 2.4/i386 initialization algorithm (still very didactical)

- 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`)

Initialization function `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)));
        ……

        }
}
```

Note!!!

- **The final PDE buffer coincides with the initial page table that maps** 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
- **Otherwise memory mapping would be lost upon any TLB miss**

The `set_pmd` macro

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

- Thia macro simply sets the value into one PMD entry
- Its input parameters are
  - the `pmdptr` pointer to an entry of PMD (the type is `pmd_t`)
  - The value to be loaded `pmdval` (of type `pmd_t`, defined via casting)
- While setting up the kernel page table, this macro is used in combination with `__pa()` (physical address) which returns an `unsigned long`
- **The latter macro returns the physical address corresponding to a given virtual address within kernel space (except for some particular virtual address ranges, those non-directly mapped)**
- Such a mapping deals with [3,4) GB virtual addressing onto [0,1) GB physical addressing

The `mk_pte_phys()` macro

```
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 result type is `pte_t`
- The result value can be then assigned to one PTE entry

Bootmem vs Memblock allocation API

- In memblock the classical “low pages” allocators have been incapsulated 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.

PAE (Physical address extension)

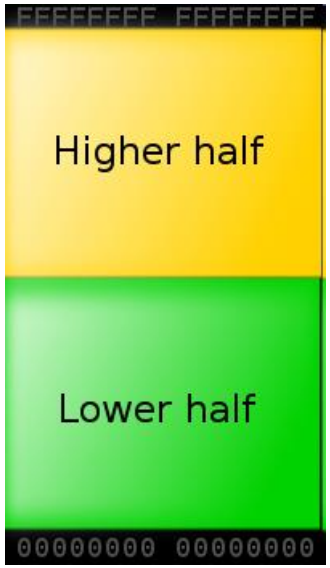
- increase of the bits used for physical addressing
- offered by more recent x86 processors (e.g. Intel Pentium Pro) which provide up to 36 bits for physical addressing
- we can drive up to 64 GB of RAM memory
- paging gets operated at 3 levels (instead of 2)
- the traditional page tables get modified 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 for 128 TB for logical addressing of individual processes and 64 TB for physical addressing

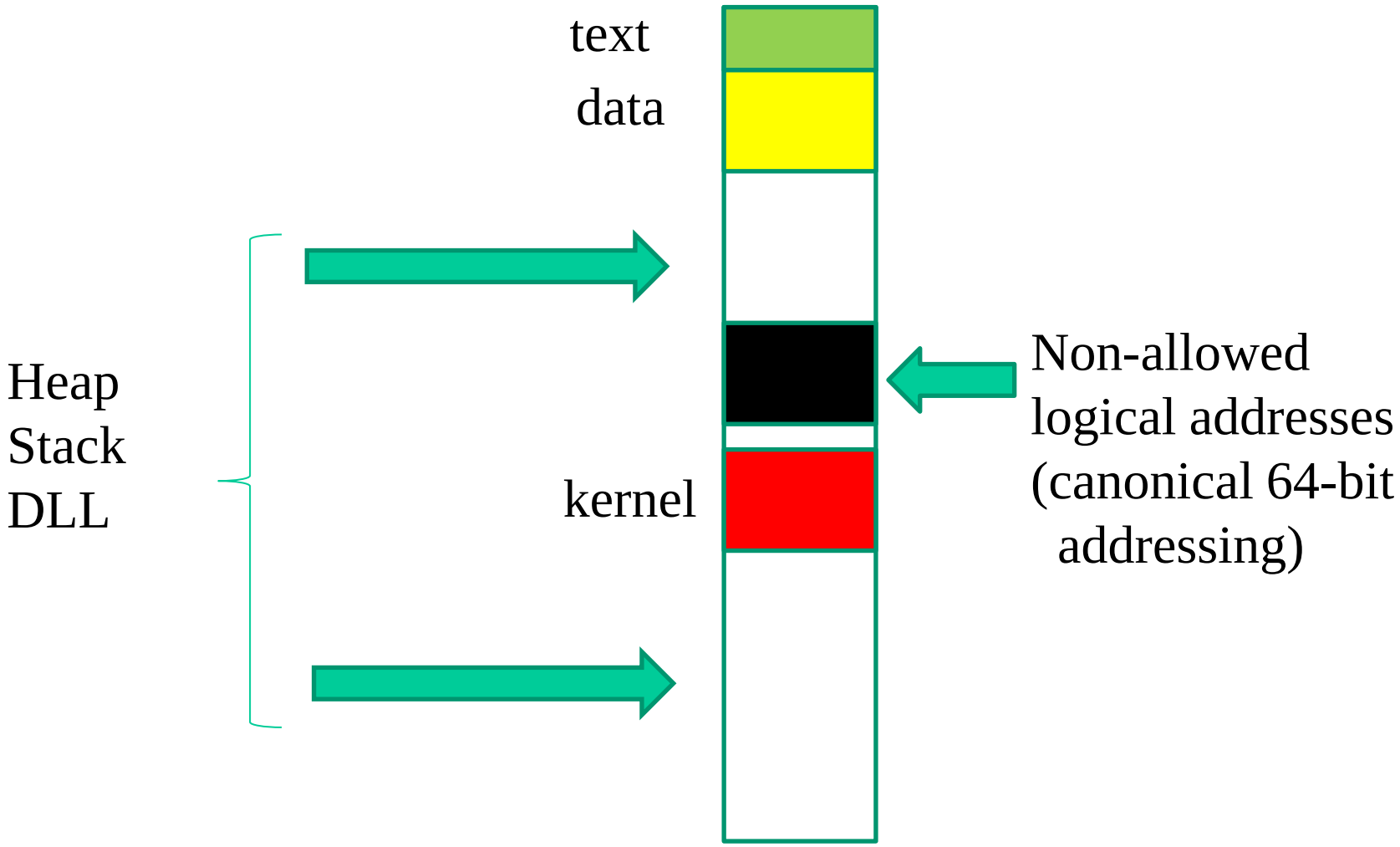
Addressing scheme

64-bit



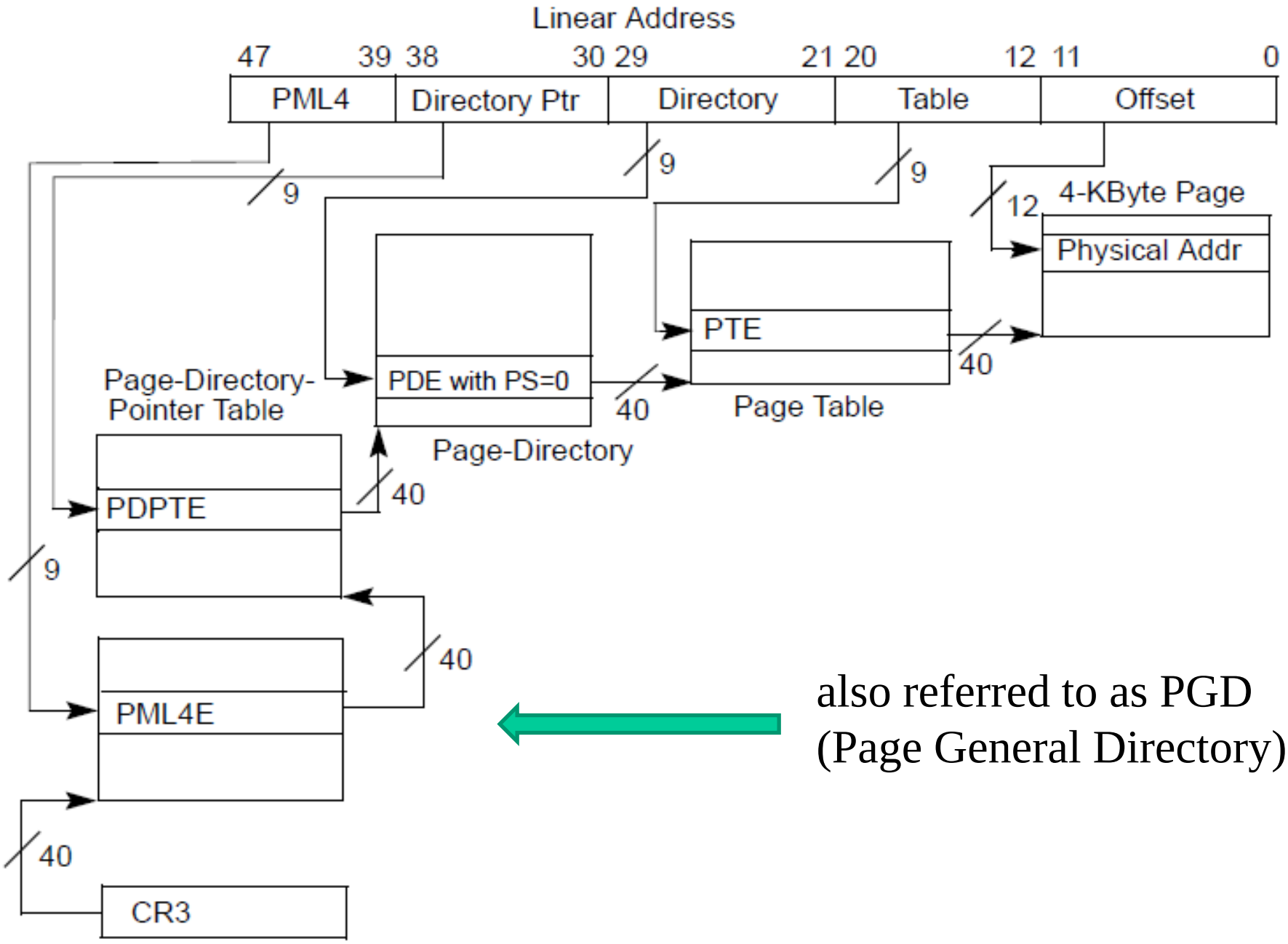
48 out of 64-bit

Linux address space on x86-64 processors



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)



also referred to as PGD
(Page General Directory)

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

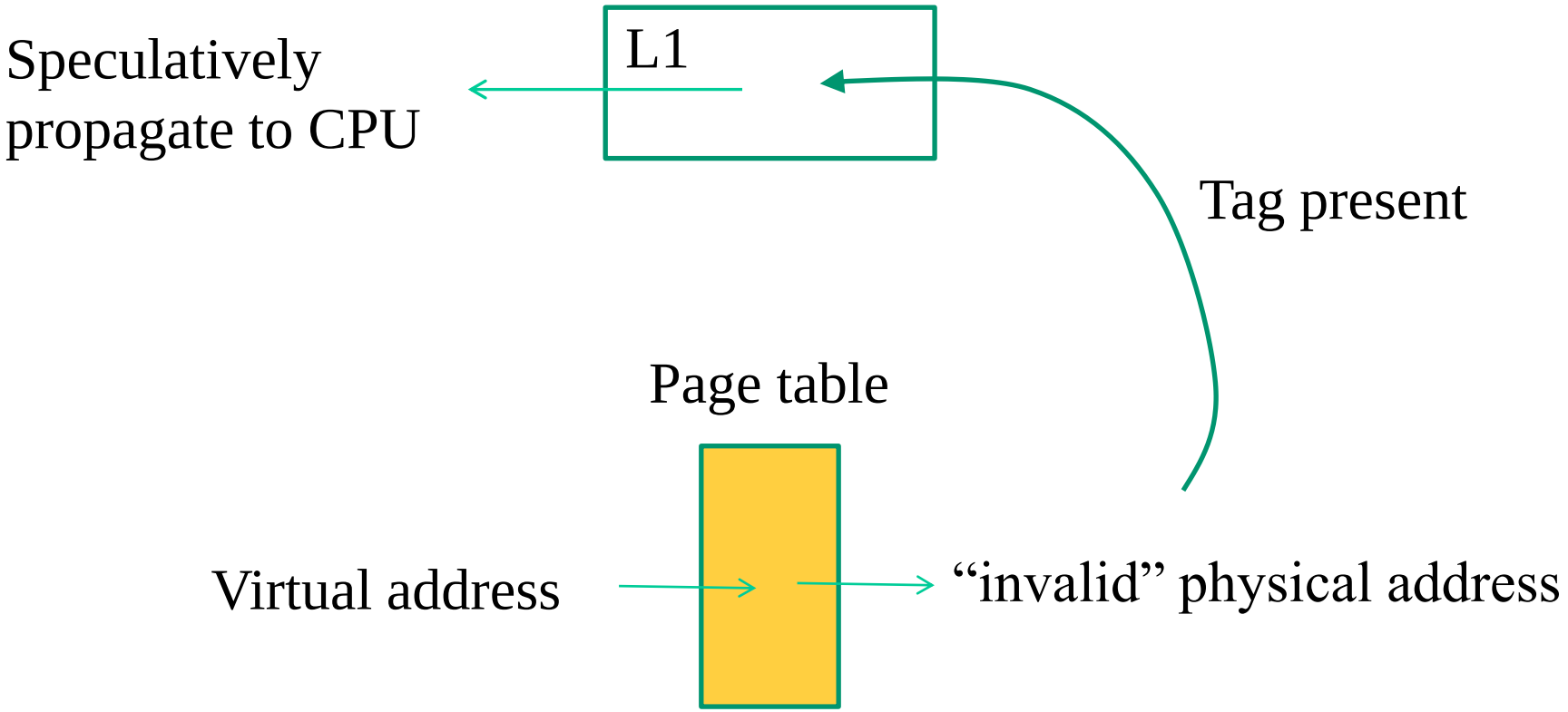
Back to speculation in the hardware

- 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: AMD, Intel ATOM, Intel Xeon PHI …

The scheme


Speculatively
propagate to CPU
L1
Tag present
Page table
Virtual address
“invalid” physical address

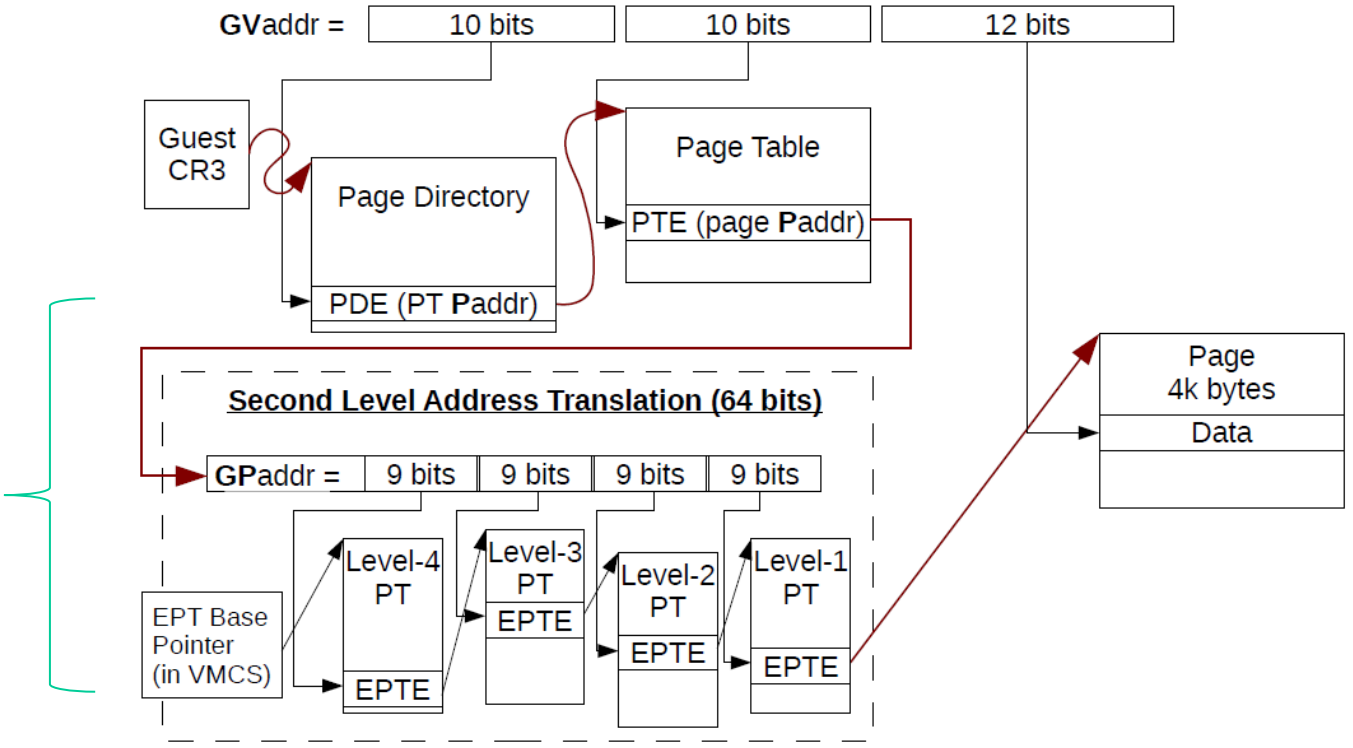
L1TF big issues

- 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 not mapping 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

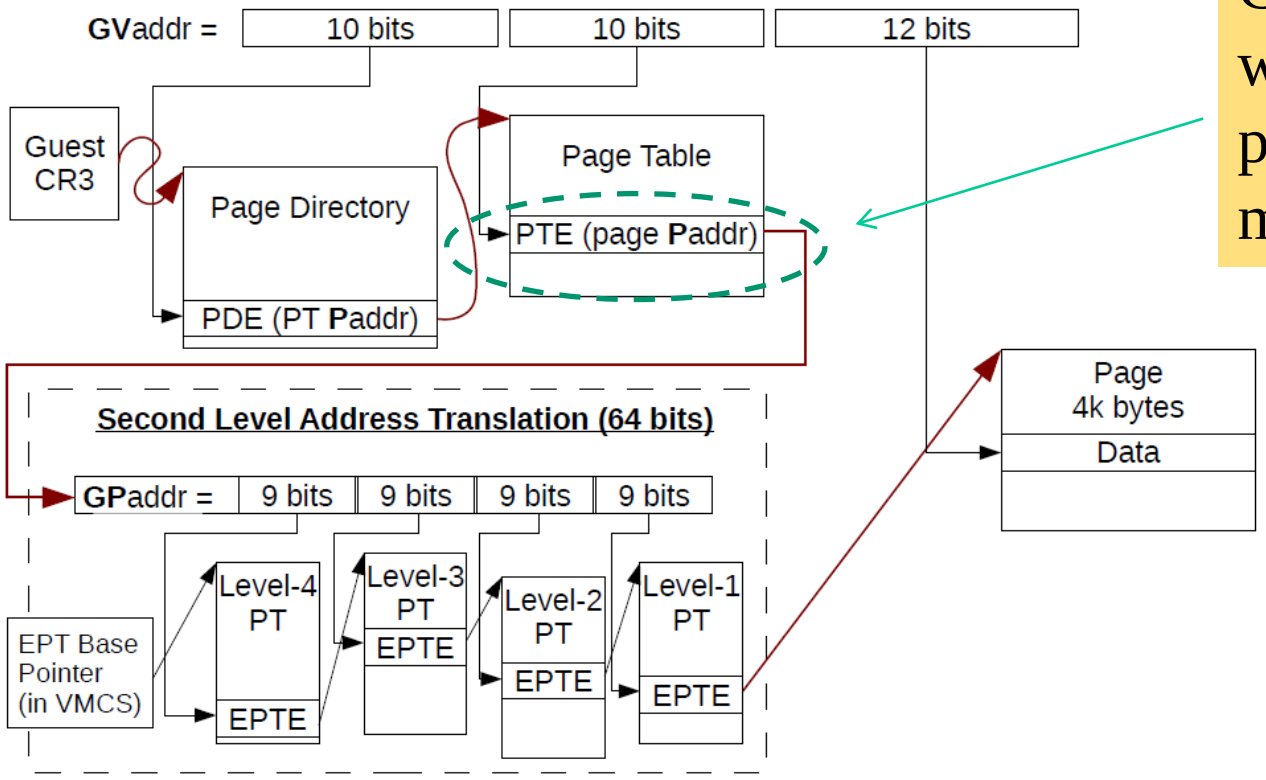
Hardware supported “virtual memory” virtualization

- 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

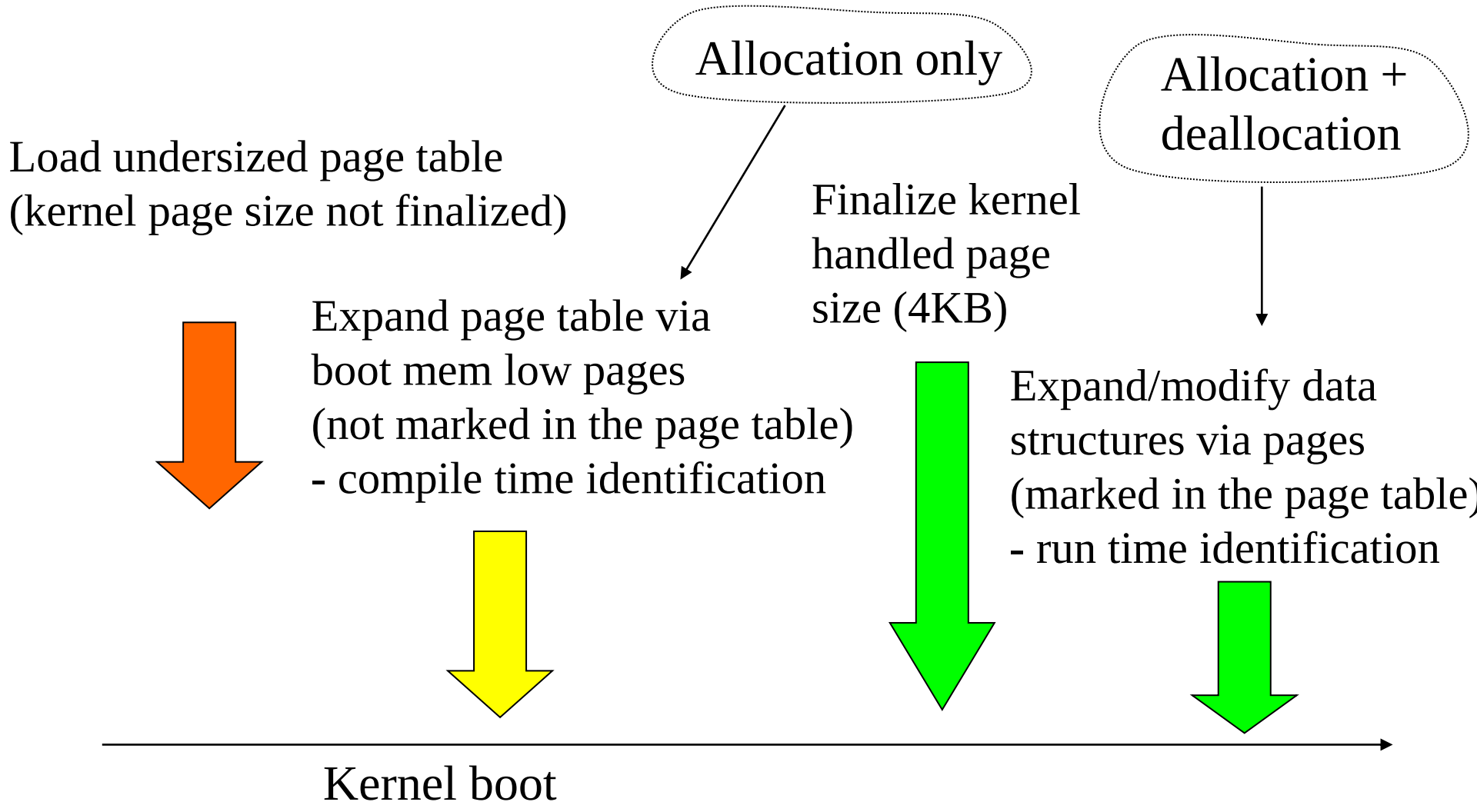


Attacking the host physical memory



Change this to whatever physical address and make the entry invalid

Reaching vs allocating/deallocating memory



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;
```

The concept of memory zones

- Historically (e.g. when reasoning on sigle NUMA-node or NUMA unaware kernels) we had 3 free lists of frames
- Hence, we had frame positioning within the following zones: DMA (DMA ISA operations), NORMAL (room where the kernel can reside), HIGHMEM (room for user data)
- With classical 32-bit address spaces the corresponding sizes were

```
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
```

Linux free list data structures

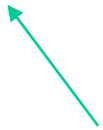
- Free lists information is kept within the `pg_data_t` data structure defined in `include/linux/mmzone.h`, and whose actual instance is `contig_page_data`, which is declared in `mm/numa.c`

```
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

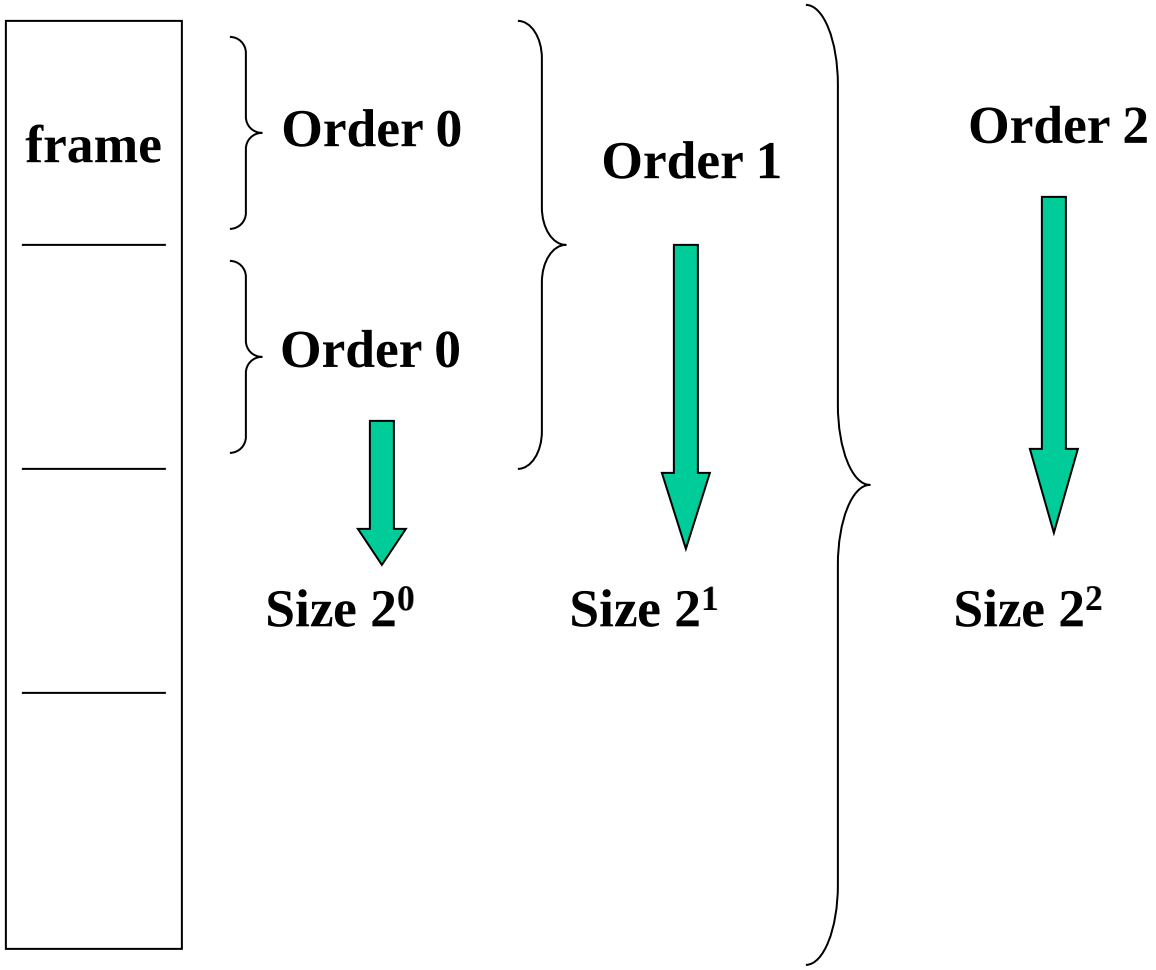
```
struct zone {
        ……
        free_area_t          free_area[MAX_ORDER];
        ……
        spinlock_t           lock;
        ……
        struct page          *zone_mem_map;
        ……

}
```

Where do we pick free memory blocks in a buddy allocator

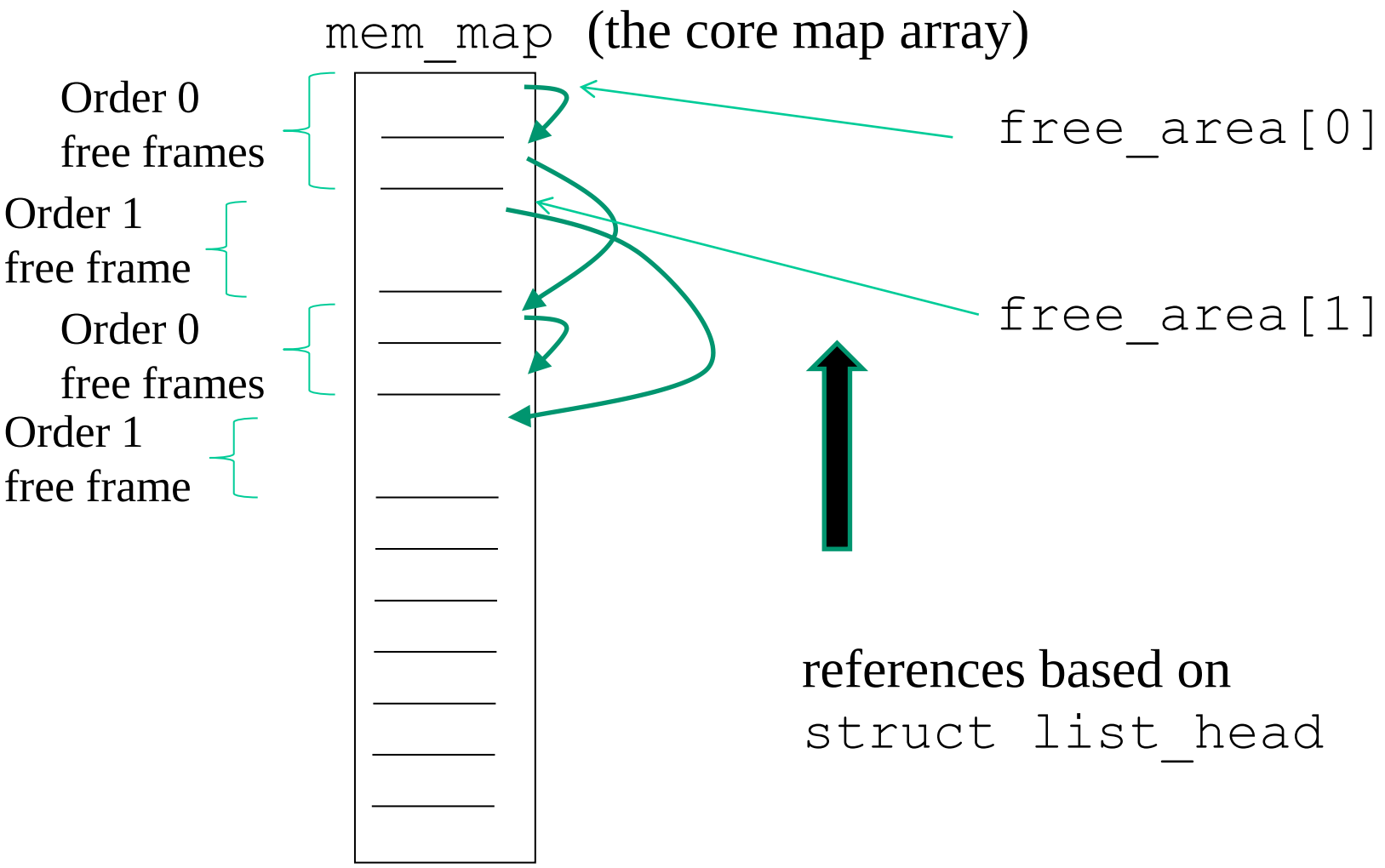


Up to 11 in recent kernel versions (it was typically 5 before)

Buddy system features

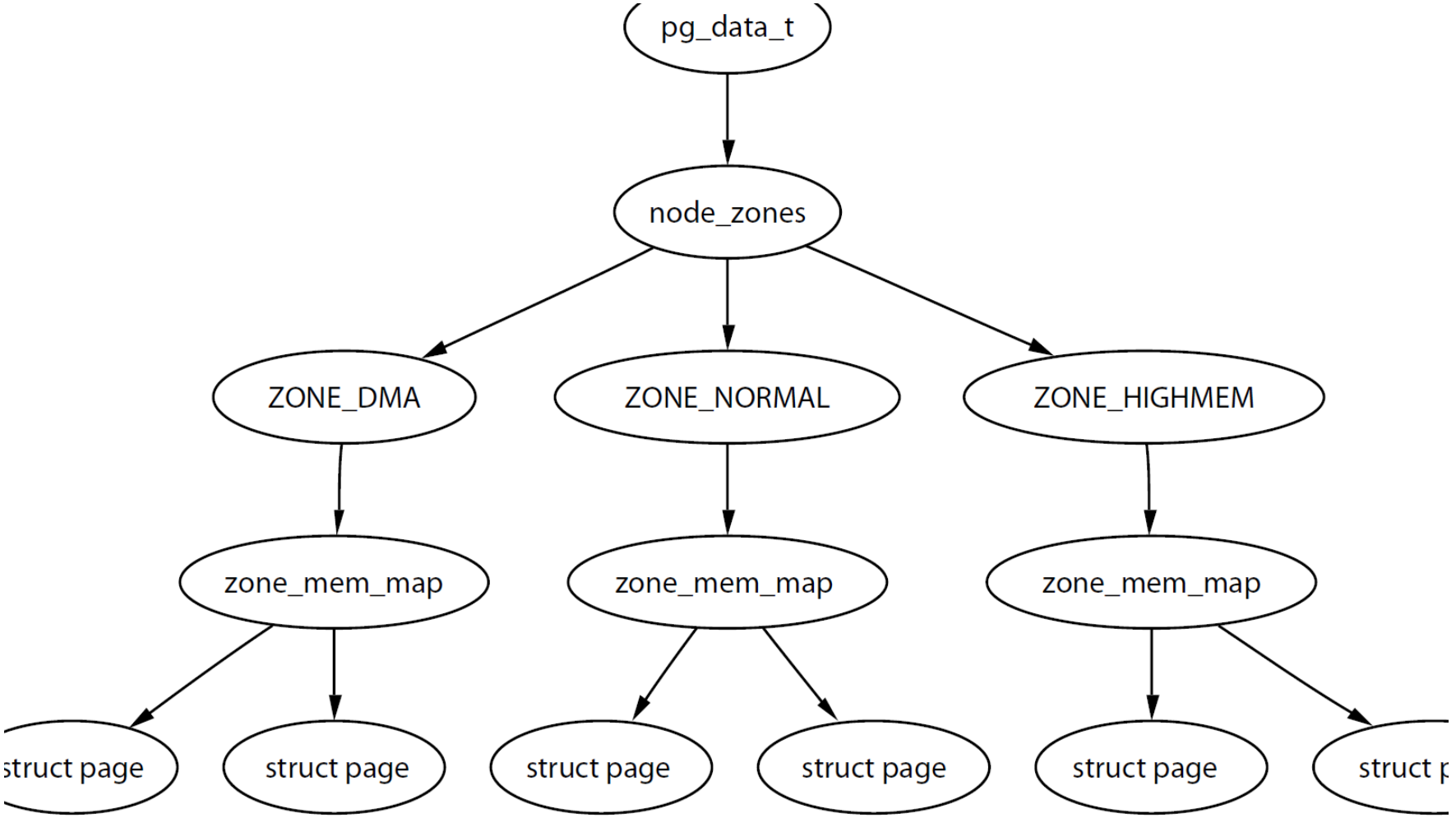


Buddy allocation vs core map vs free list



Recall that spinlocks are used to manage this data structure

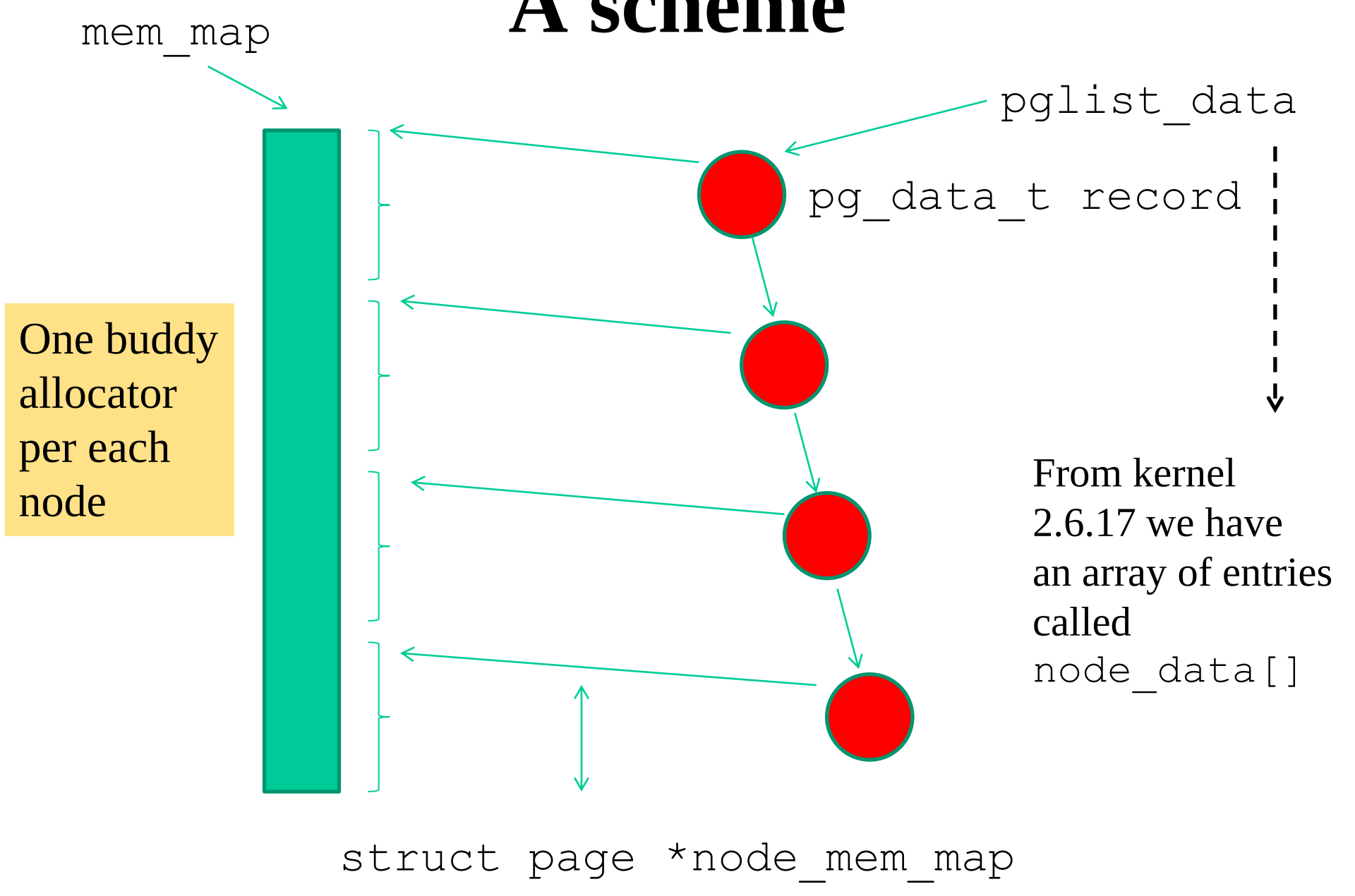
A scheme (picture from: Understanding the Linux Virtual Memory Manager – Mel Gorman)



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 `pgdat_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
pglist_data
pg_data_t record
One buddy allocator per each node
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

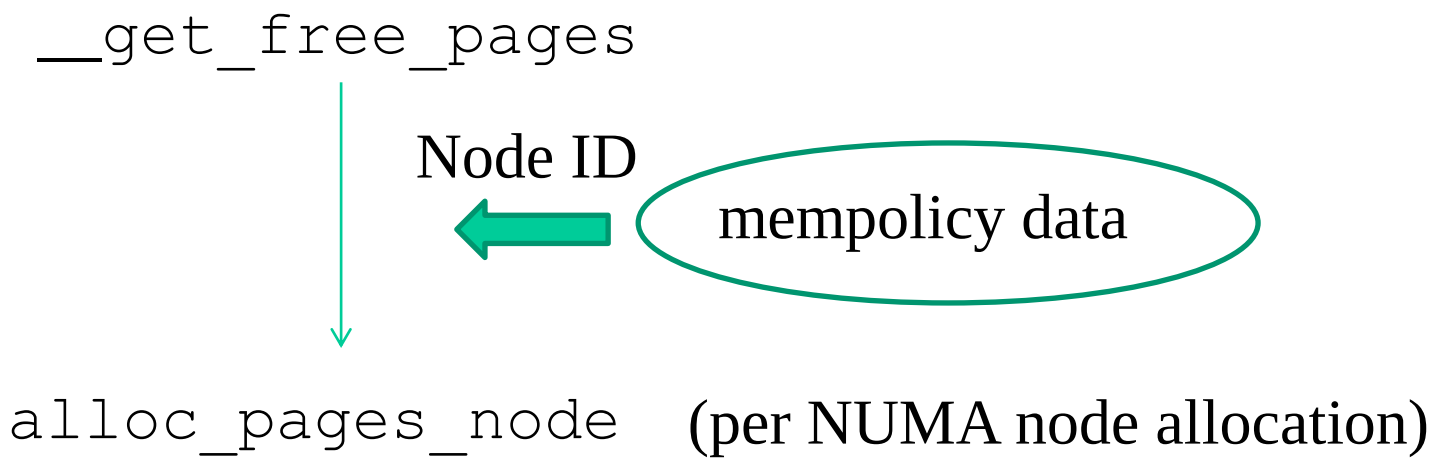
- All the buddy-system API functions return virtual addresses of direct mapped pages
- We can therefore directly discover the position in memory of the corresponding frames
- Also, memory contiguousness is guaranteed for both virtual and physical addresses

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:



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

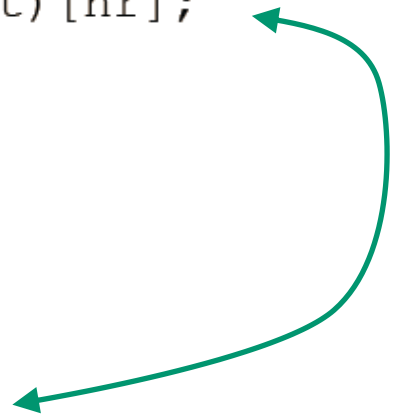
Actual quicklists

- 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()`
- In very latest versions of the Linux kernel, pre-reserving is done at the buddy allocator API

Quicklist API and algorithm

```
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!!

Logical/Physical address translation for kernel directly mapped memory

We can exploit the below macros

`virt_to_phys(unsigned int addr)`

(in `include/asm-i386/io.h`)

`phys_to_virt(unsigned int addr)`

(in `include/asm-i386/io.h`)

`__pa()`

`__va()`

} In generic kernel versions

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

- The prototypes are in `linux/malloc.h`
- 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 related data within the same chunk)
  - Fast allocation/deallocation support

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

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 direct 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

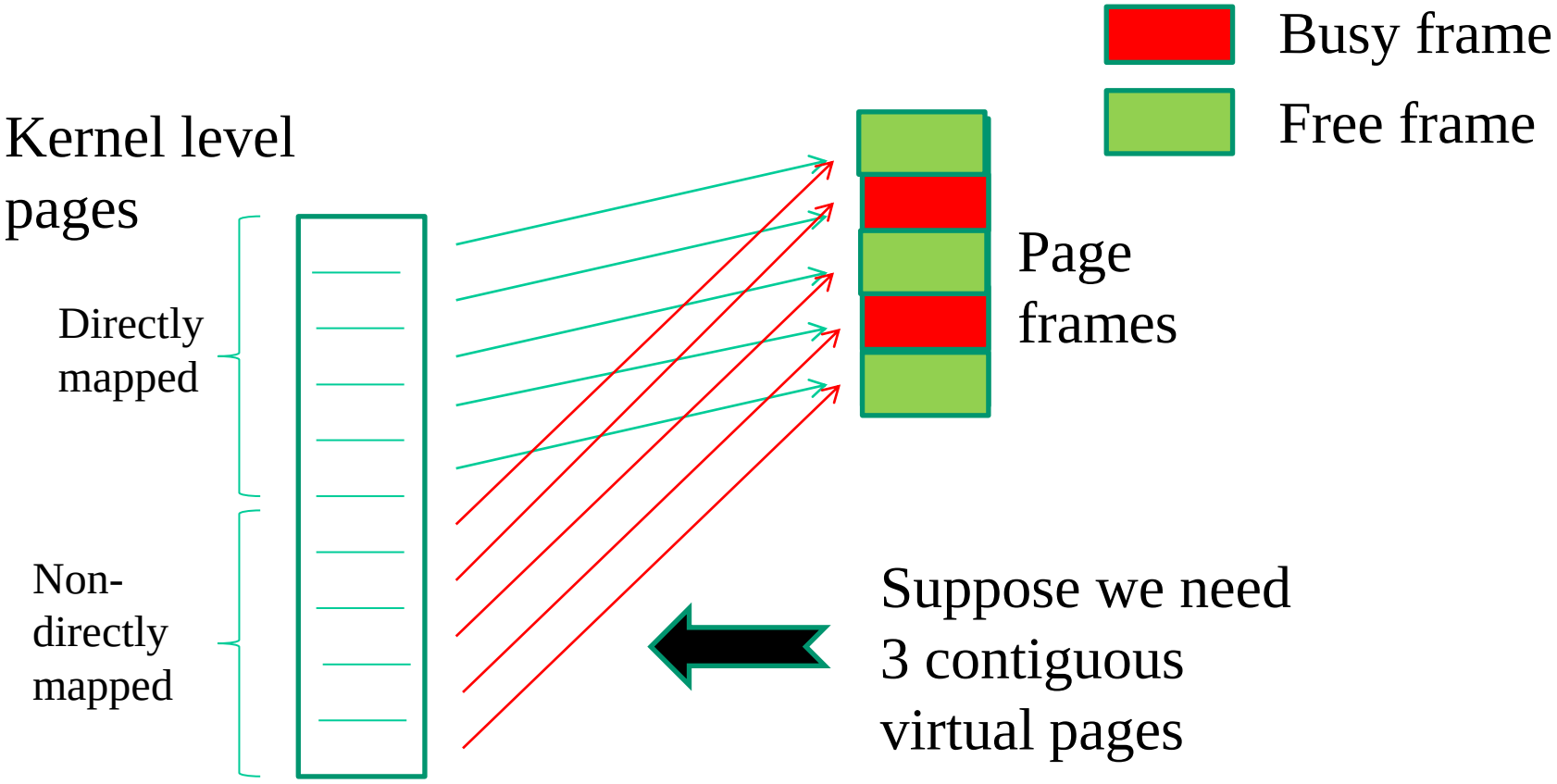
- Allocation size:
  - 128 KB for `kmalloc` (cache aligned)
  - 64/128 MB for `vmalloc`

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

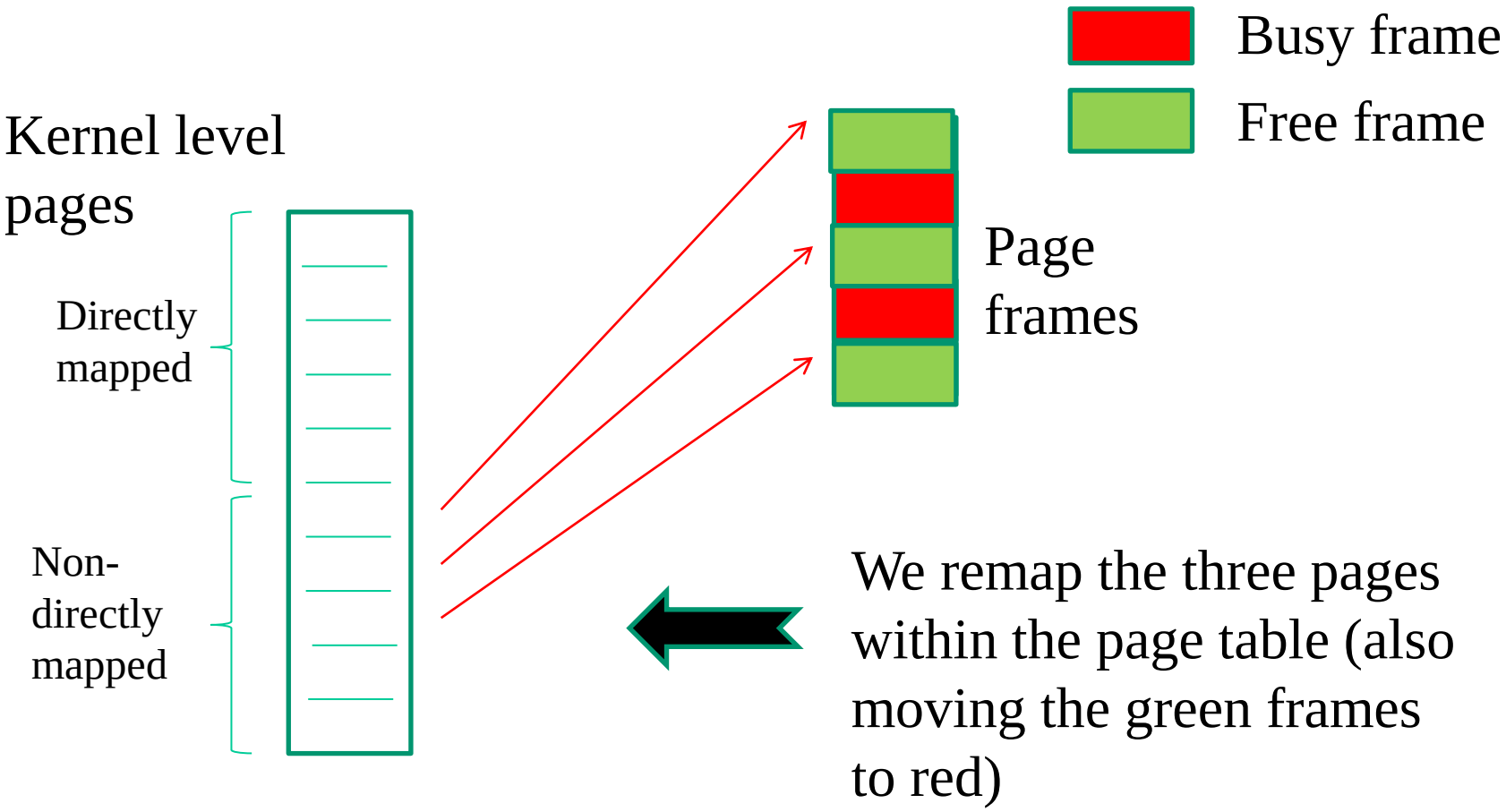
- Effects on TLB
  - None for `kmalloc`
  - Global for `vmalloc` (transparent to `vmalloc` users)

`vmalloc` operations (i)

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



`vmalloc` operations (ii)



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, all the other cores need to observe the update mapping
- 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)
```

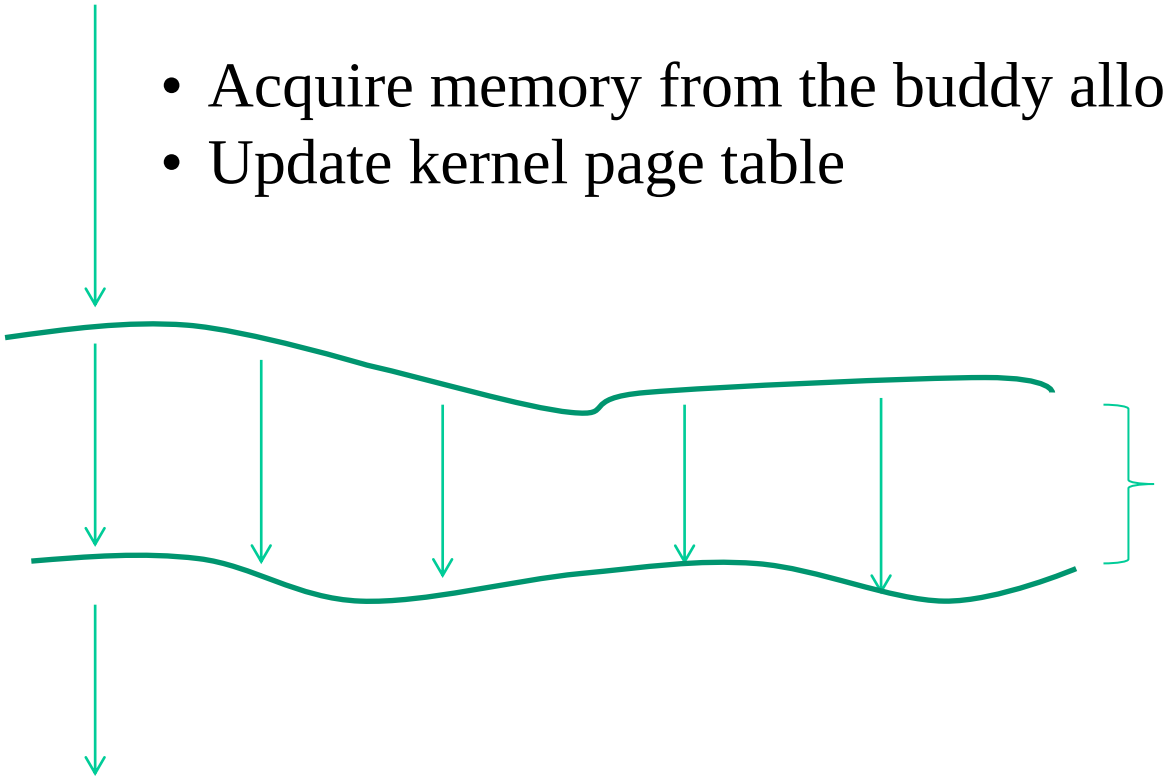
- This flushes the entire TLB **on all processors running in the system**, which makes it the most expensive TLB flush operation
- After it completes, all modifications to the page tables will be visible globally
- This is required after the kernel page tables, which are global in nature, have been modified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

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

return

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
- In some architectures (e.g. MIPS), this will need to be performed for all processors, but usually it is confined to the local processor
- 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)**