MS degree in Computer Engineering
University of Rome Tor Vergata
Lecturer: Francesco Quaglia

Topics:

1. The very base on boot vs memory management
2. Memory ‘Nodes’ (UMA vs NUMA)
3. Paging support in x86
4. Boot and steady state behavior of the memory system in LINUX
5. Kernel level memory allocation/deallocation services

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)

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

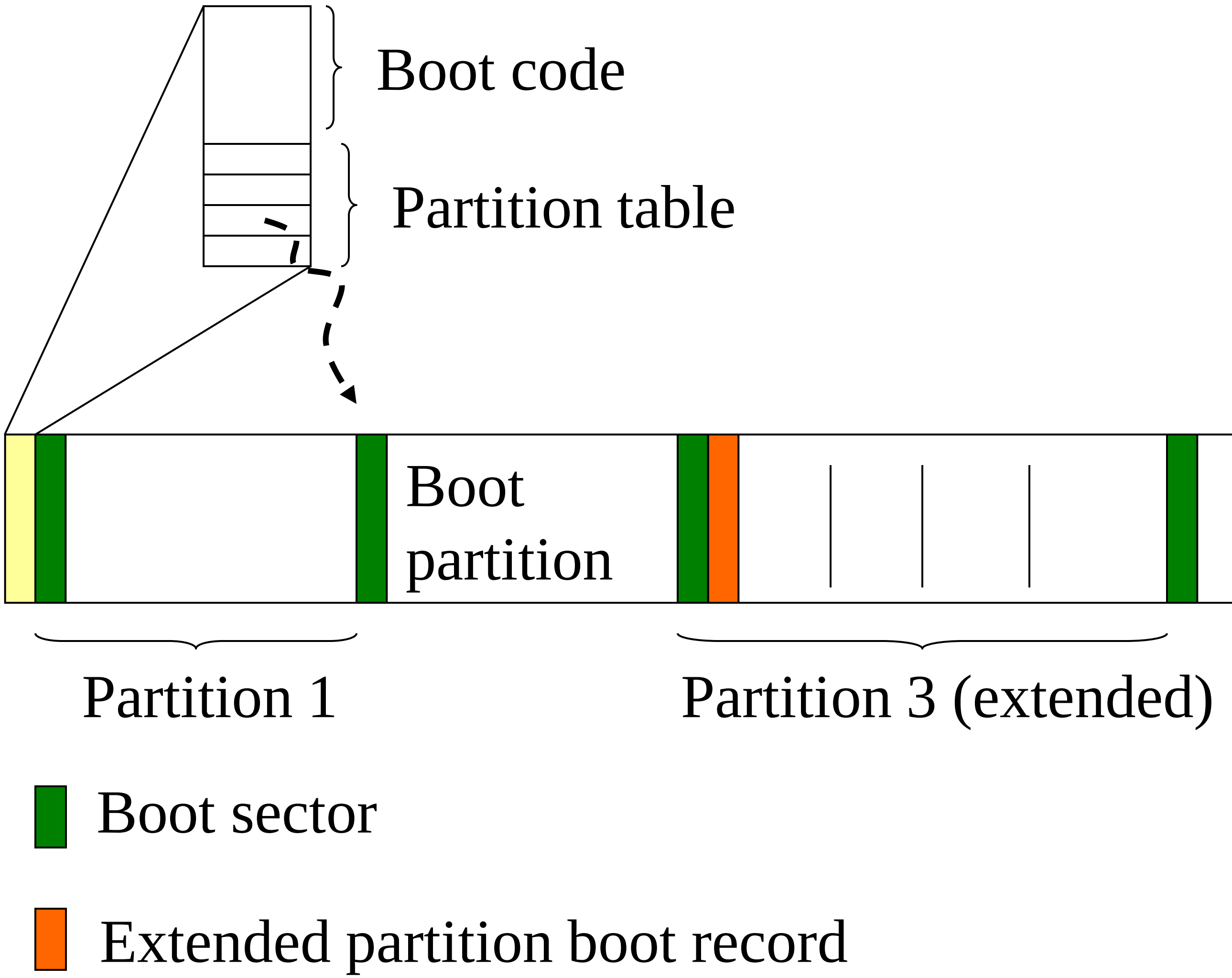
Bootsector on x86

- The first device sector keeps the so called **master boot record** (MBR)
- This sector keeps executable code and a 4/8-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 x86 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

An example scheme



Historical LINUX bootsector organization for i386

- The historical bootsector code for LINUX (i386) is within the kernel file `arch/i386/bootsect.S` (it is no more used in case boot operates via LILO or GRUB bootloaders)
- This code loads `arch/i386/bootsetup.S` and the kernel image in memory
- The code within `arch/i386/bootsetup.S` gets launched for initializing the architecture (e.g. the processor initial state for the actual kernel boot)
- This code ultimately gives control to the initial kernel image

Current LINUX bootsector organization on x86

- Similar steps are executed via, e.g., GRUB on generic x86 machines
- The machine setup code ultimately passes control to the initial kernel image
- This kernel image executes starting from the `start_kernel()` in the `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 highly configurable!

What about boot on multi-core 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

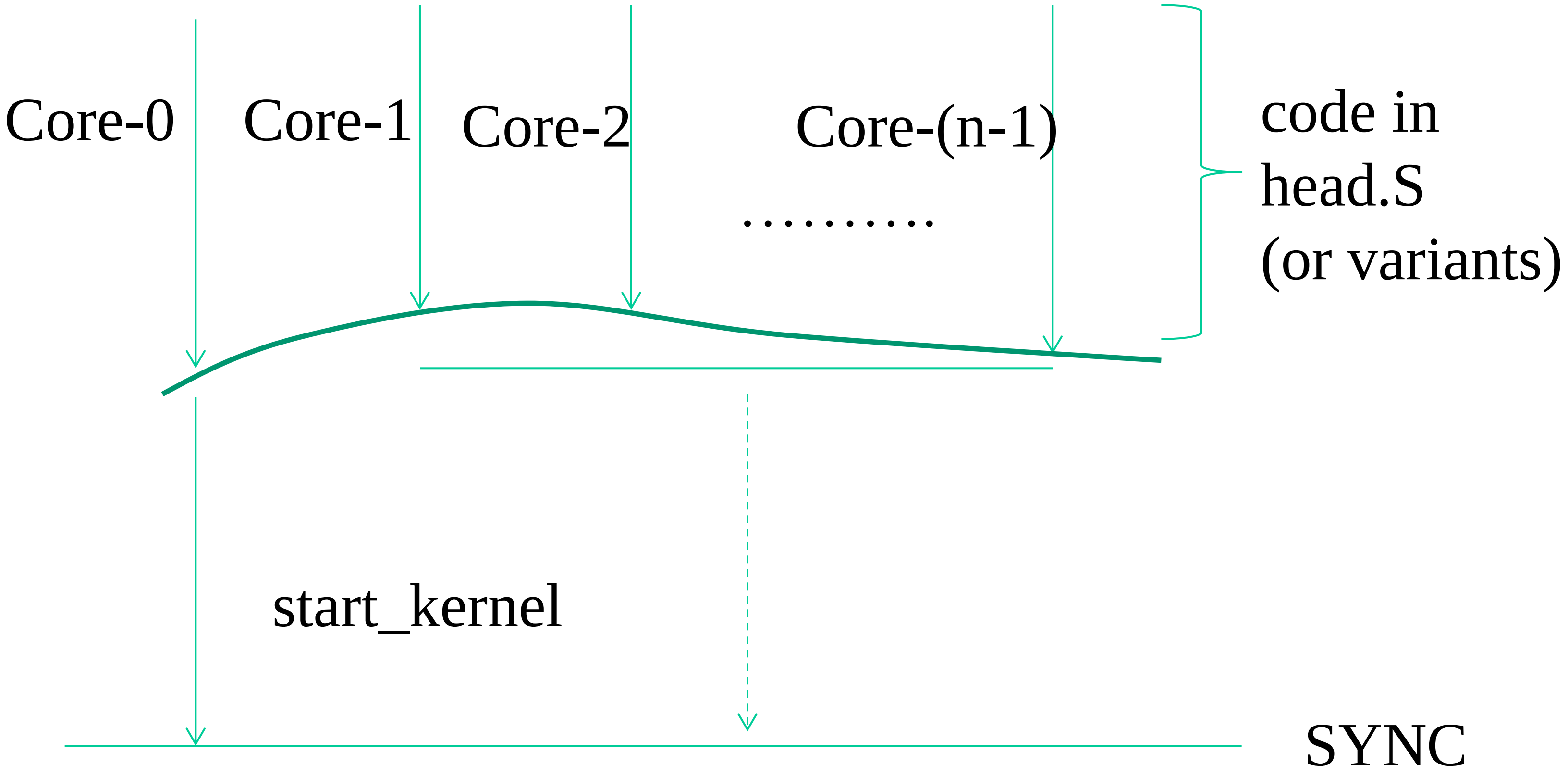
x86 Instruction Set Reference

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



An example `head.S` code snippet: triggering 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
*/
```

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)

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 (kernel is resident)

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

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 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 …….

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

Setup of the kernel page table

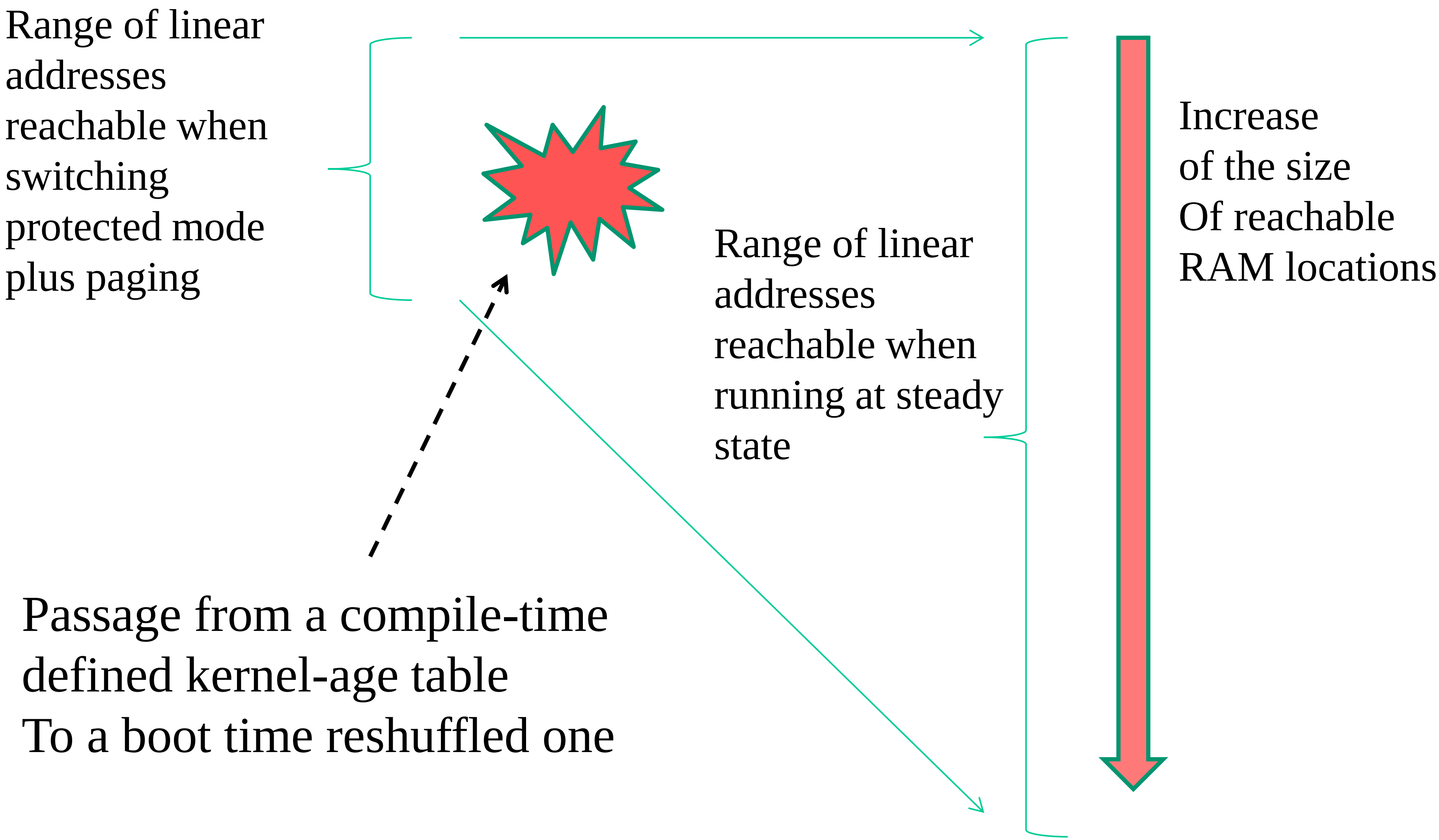
- This setup takes place via `setup_arch()` which is present in `arch/XX/kernel/setup.c`
- The main function invoked here is a configuration specific version of `paging_init()`
- As an example, for i386 processors this function is specified in `arch/i386/mm/init.c`
- Clearly, new generations of processors provide different (more powerful) paging support and clearly a different page table structure
- We will have a tour starting from i386 and then finally reaching x86-64 processors (via incremental differences)

Objectives of the kernel page table setup

- These are basically two:
  - ✓ Allowing the kernel software to use virtual addressed 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

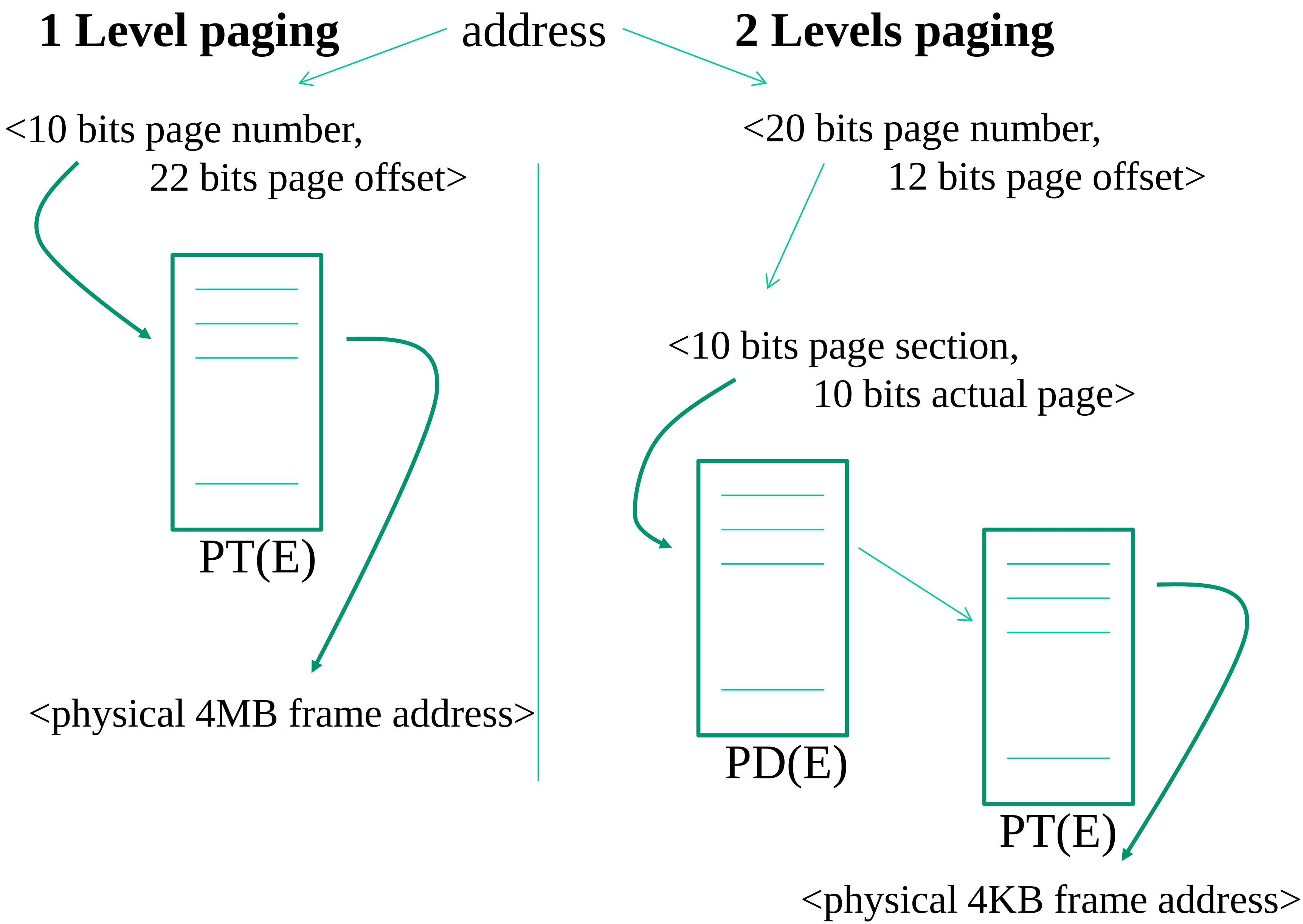
A scheme


Range of linear addresses reachable when switching protected mode plus paging
Range of linear addresses reachable when running at steady state
Increase of the size Of reachable RAM locations
Passage from a compile-time defined kernel-age table
To a boot time reshuffled one

Virtual memory vs boot sequence (2.4 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
- 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 is 4 GB
  - 3 GB are within user level segments
  - 1 GB is within kernel level segments

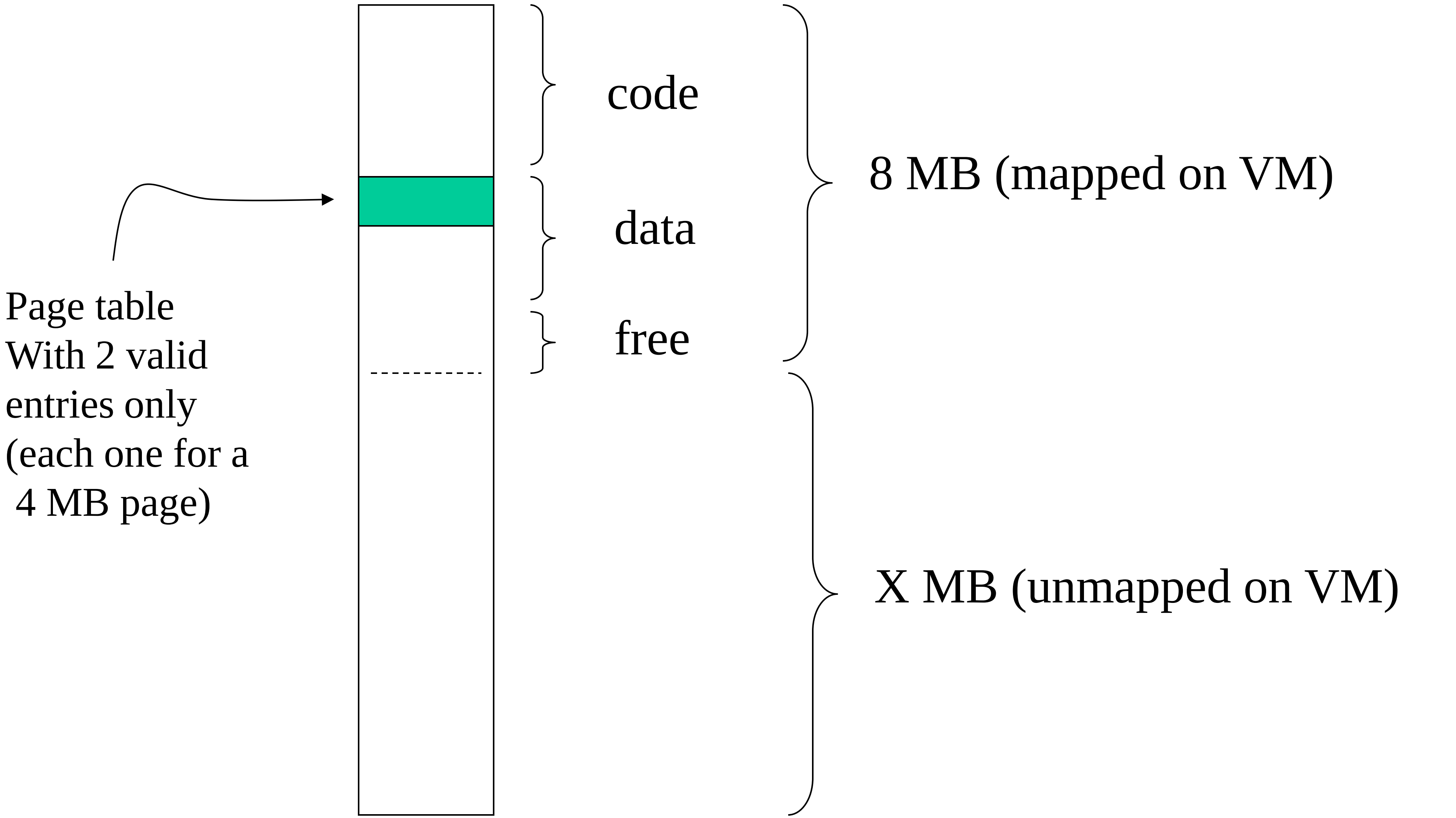
Details on the page table structure in i386 (i)



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 for 2.4 kernels



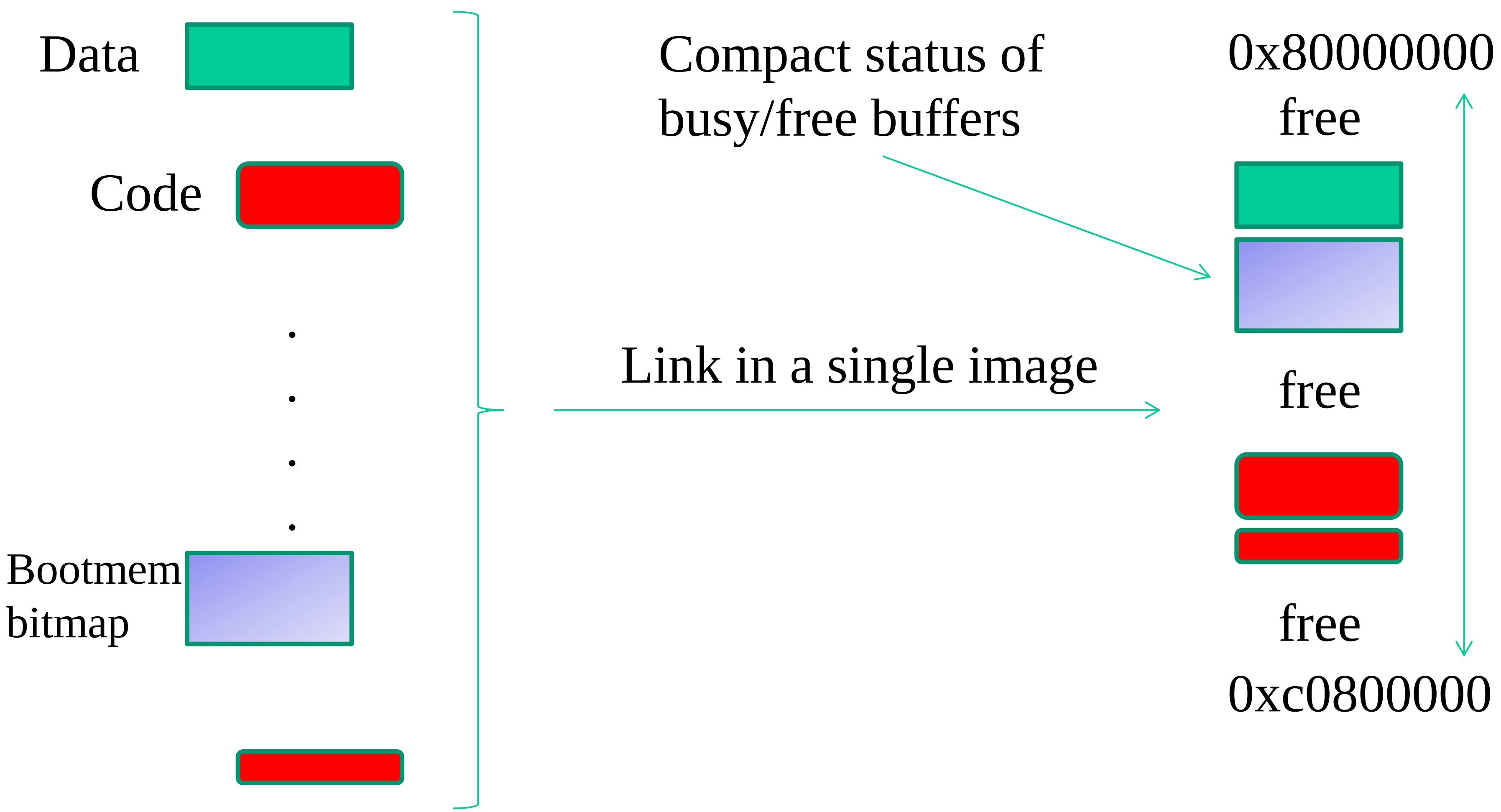
Actual issues to be tackled

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 manageable physical memory
3. We need to re-organize the page table in two separate levels
4. So we need to determine ‘free buffers’ within the already reachable memory segment to initially expand the page table
5. We cannot use memory management facilities other than paging (since core maps and free lists are not at steady state)

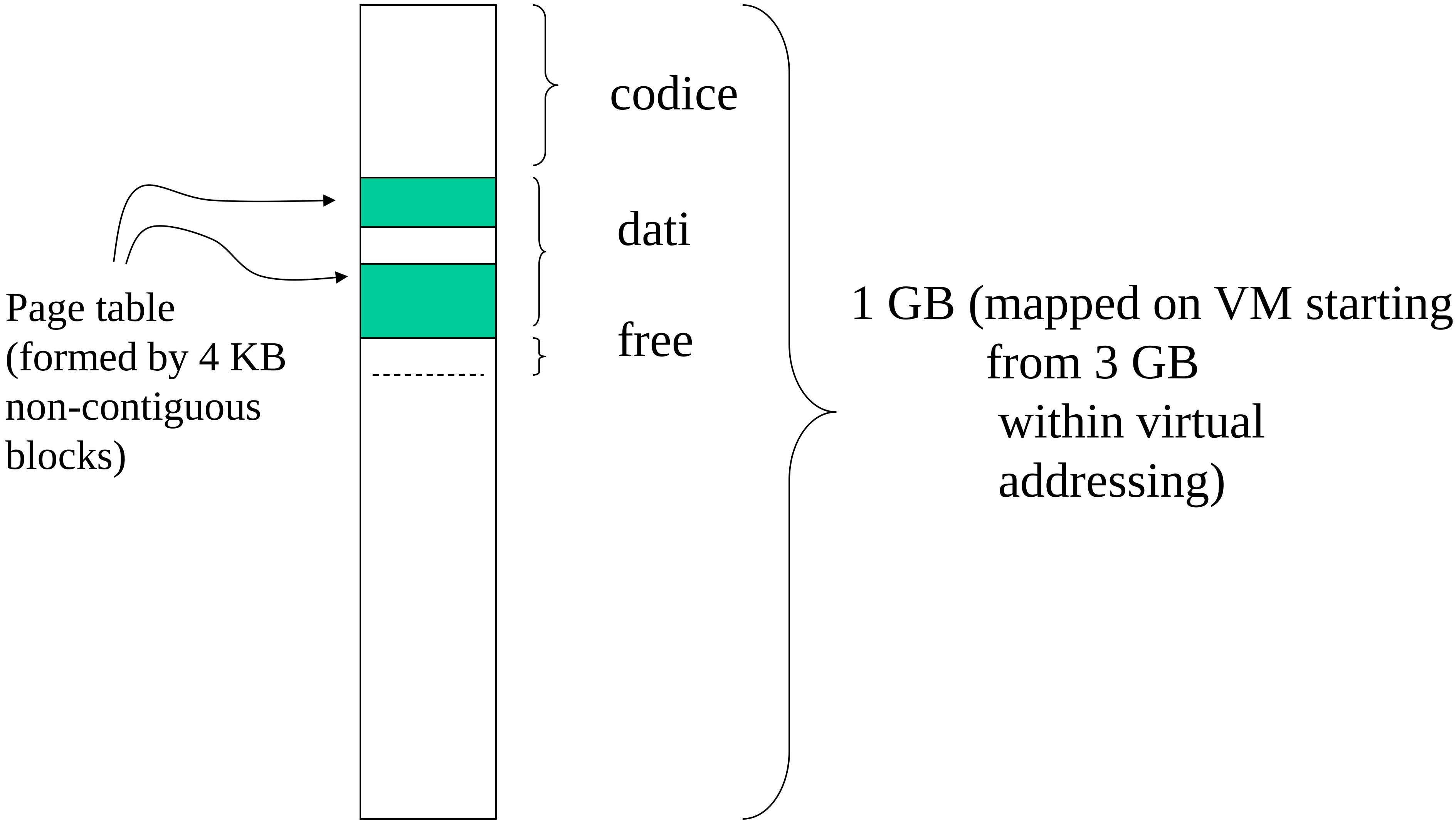
Back to the concept of *bootmem*

1. Memory occupancy and location of the initial kernel image is determined by the compile/link process
2. 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 alligned areas
6. This subsystem is in charge of handling `_init` marked functions in terms of final release of the corresponding buffers

An exemplified picture of bootmem (i386/kernel 2.4)

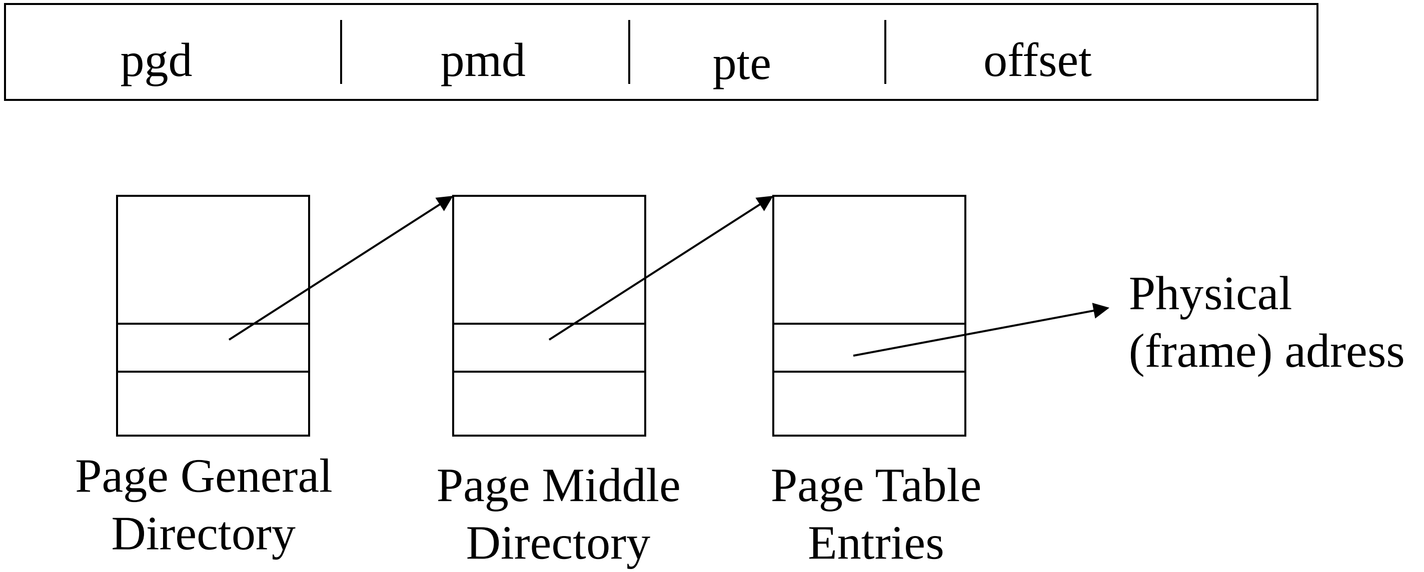


Page table collocation within physical memory



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)
- <u>**Such a dicotomy is solved by setting null**</u> the `pmd` field, which is proper of LINUX, and mapping
  - ➢`pgd` LINUX on `pde` i386
  - ➢`pte` LINUX on `pte` i386

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;
```

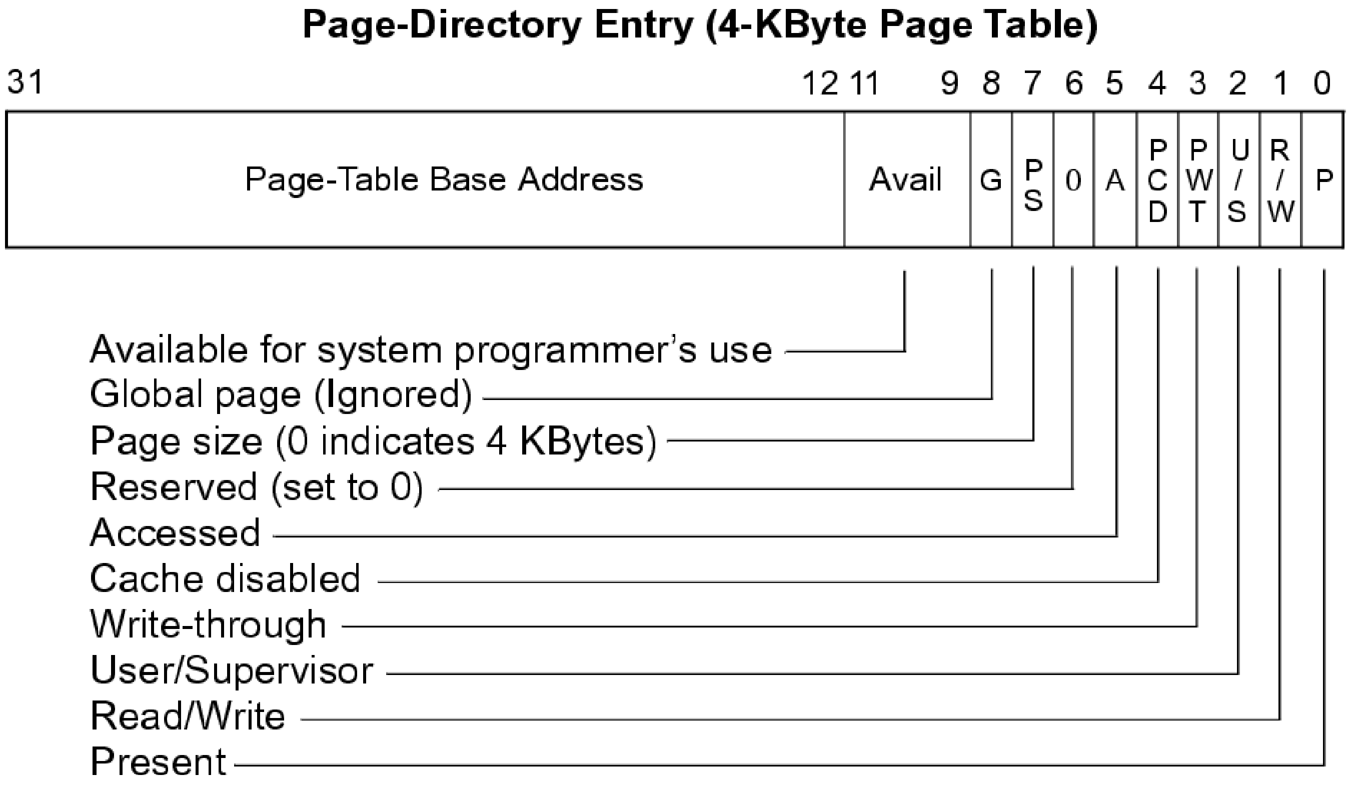
Debugging

- 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

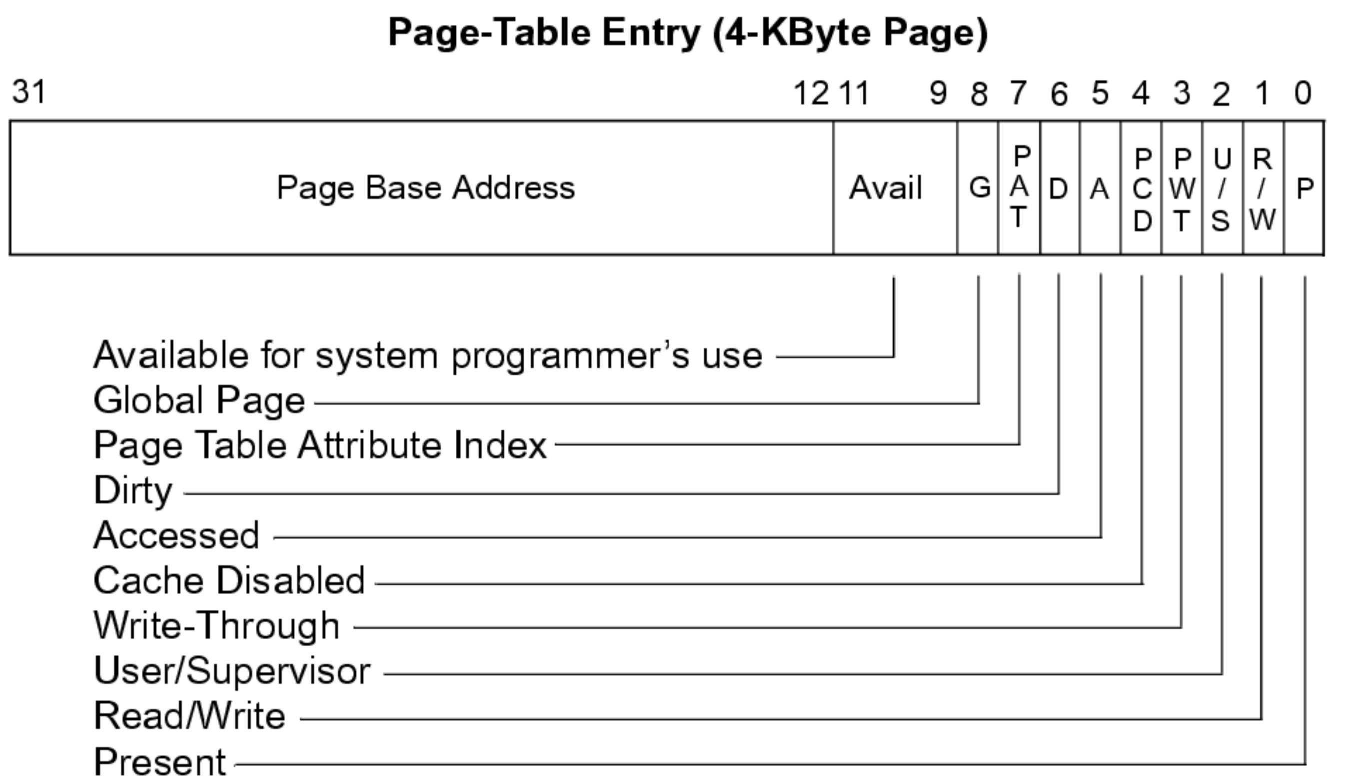
  ```
  typedef unsigned long pgd_t;
  typedef unsigned long pte_t;
  pgd_t x; pte_t y;
  ```

  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



i386 PTE entries



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 TLP 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_PWT          0x008
#define _PAGE_PCD          0x010
#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 the 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 on IA-32 processors

```
#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);
}
```

Low level “pages”

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

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

Finalize kernel handled
page size (4KB)

Kernel boot

Kernel page table initialization

- As said, the kernel PDE is accessible at the virtual address kept by `swapper_pg_dir` (now `init_level4_pgt` on x86-64/kernel 3 or `init_top_pgt` on x86-64/kernel4)
- 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 upon which the LINUX memory system boot lies on

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`)

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)**
- 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

PAE (Physical address extension)

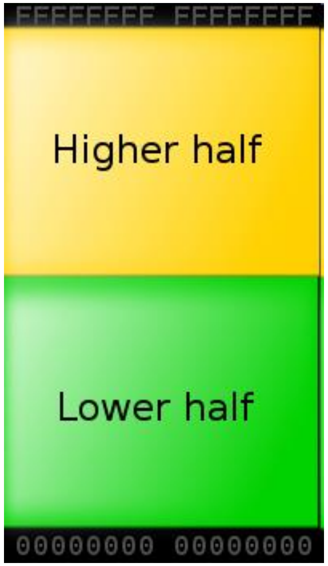
- increase of the bits used for physical addressing
- offered by more recent x86 processors (e.g. Intel Pentium Pro) which provide up to 36 bits for physical addressing
- we can drive up to 64 GB of RAM memory
- paging gets operated at 3 levels (instead of 2)
- the traditional page tables get modified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

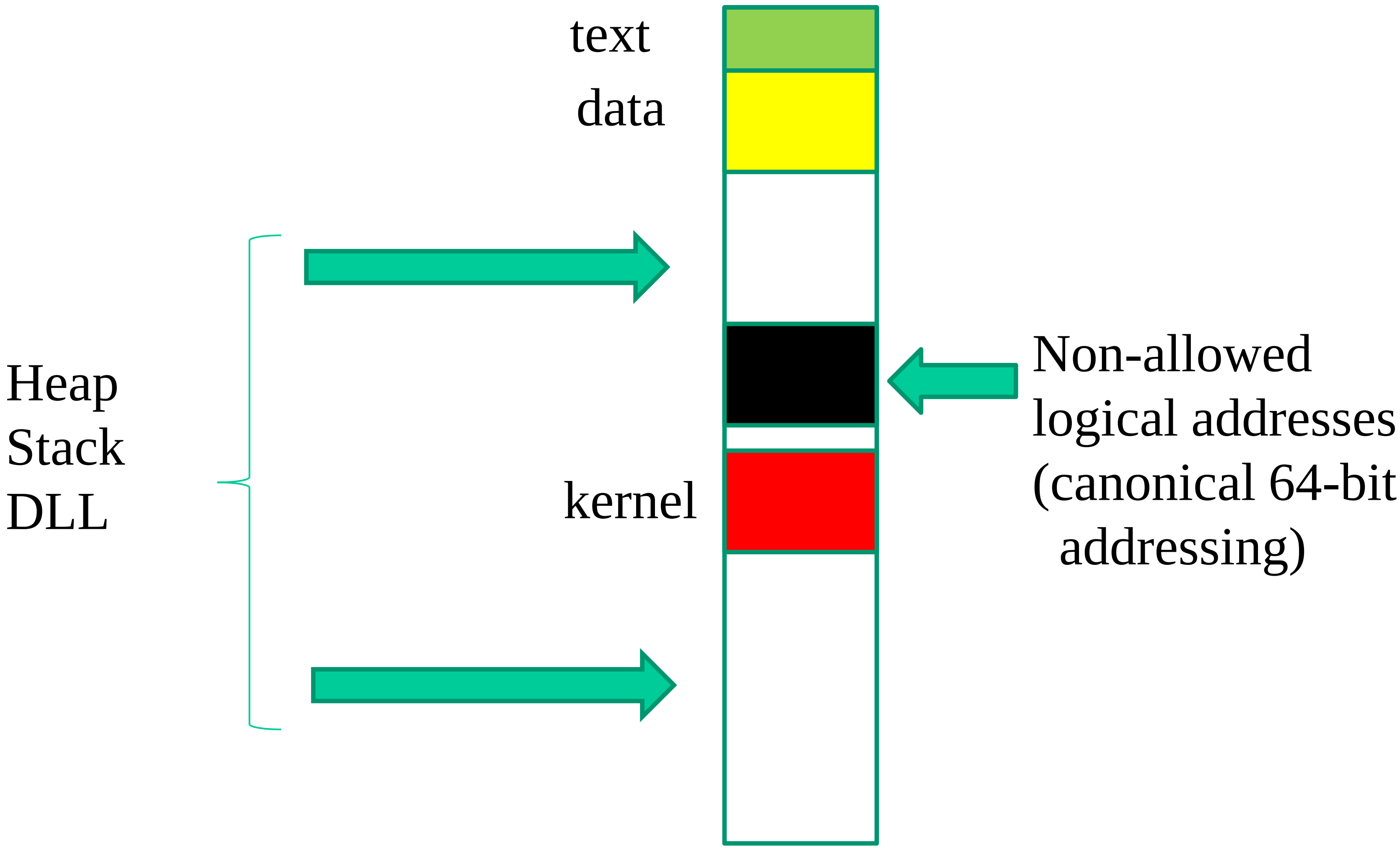
Addressing scheme

64-bit



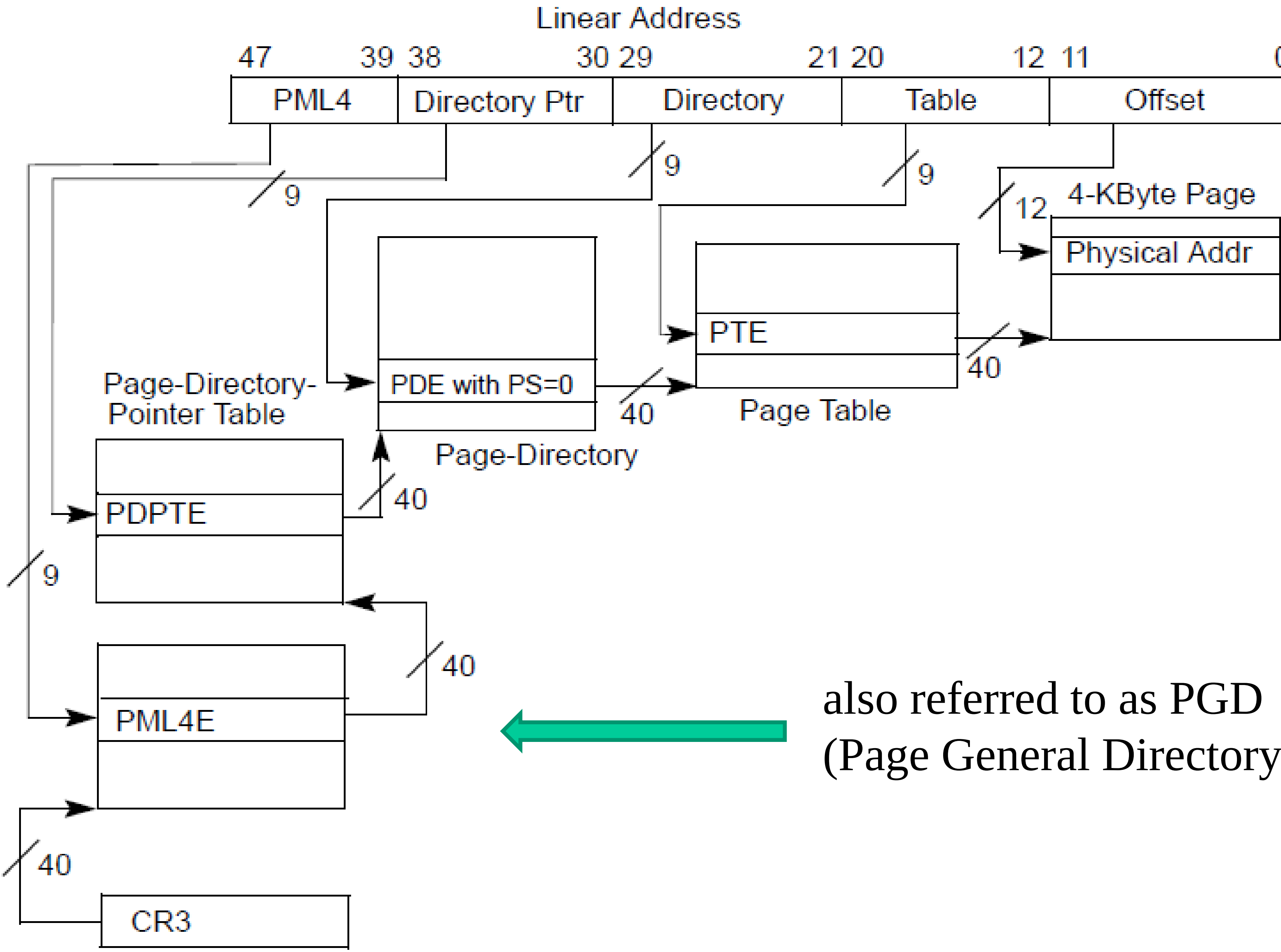
48 out of 64-bit

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 … 52	51 … M[1]	M-1 …	…	12	11 … 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

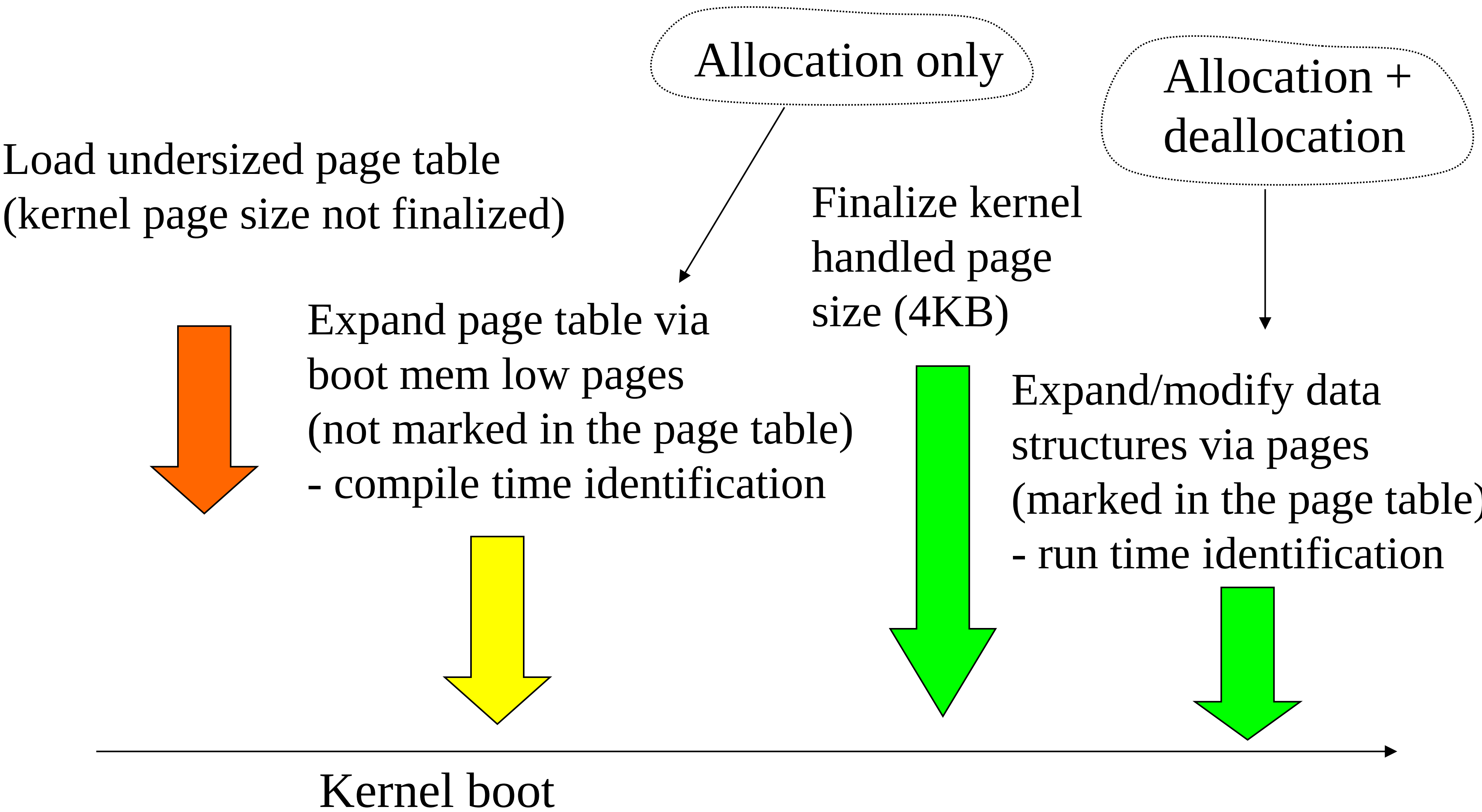
Table 4-19. Format of an IA-32e Page-Table Entry that Maps a 4-KByte Page

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)

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)

Reaching vs allocating/deallocating memory



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:

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

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

Fields

- Most of the fields are used to keep track of the interactions between memory management and other kernel sub-systems (such as I/O)

- Memory management proper fields are

    - `struct list_head list` (whose type is defined in `include/linux/lvm.h`), which is used to organize the frames into free lists
    - `atomic_t count`, which counts the virtual references mapped onto the frame (it is managed via atomic updates, such as with LOCK directives)
    - `unsigned long flags`, this field keeps the status bits for the frame, such as:

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

Core map initialization (i386/kernel 2.4)

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

Free list organization: single NUMA zone – or NUMA unaware – protected mode case (e.g. kernel 2.4)

- we have 3 free lists of frames, depending on the frame positioning within the following zones: DMA (DMA ISA operations), NORMAL (room where the kernel can reside), HIGHMEM (room for user data)
- The corresponding defines are in `include/linux/mmzone.h`:

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

- The corresponding sizes are usually defined as

```
/* 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
 processes */
```

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`

Field of interest

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

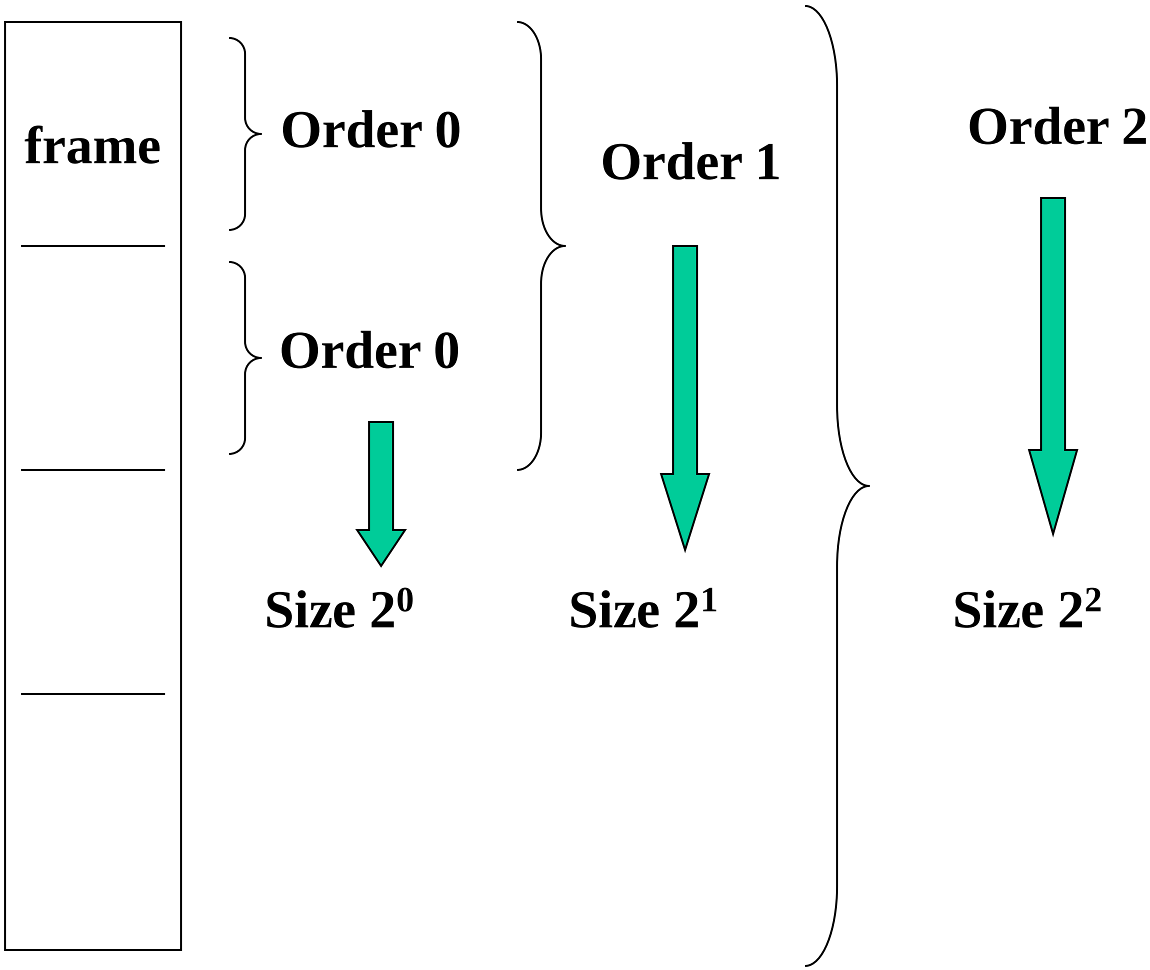
- the `zone_t` type is defined in `include/linux/mmzone.h` as follows

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

Fields of interest

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

Buddy system features



- `free_area_t` is defined in the same file as

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

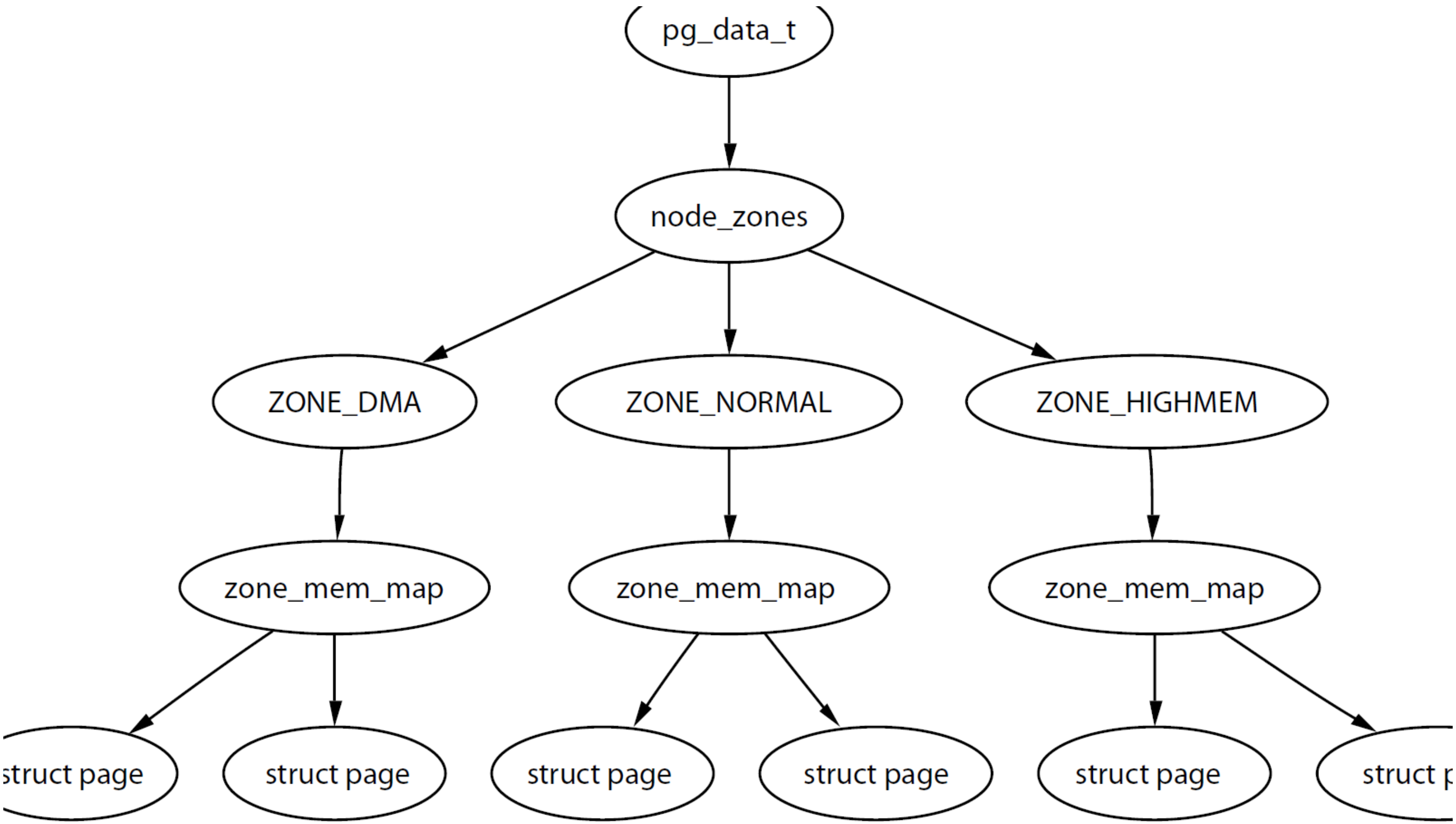
Buddies fragmentation state

- where

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

- overall, any element of the `free_area[]` array keeps
  - A pointer to the first free frame associated with blocks of a given order
  - A pointer to a bitmap that keeps fragmentation information according to the the “buddy system” organization

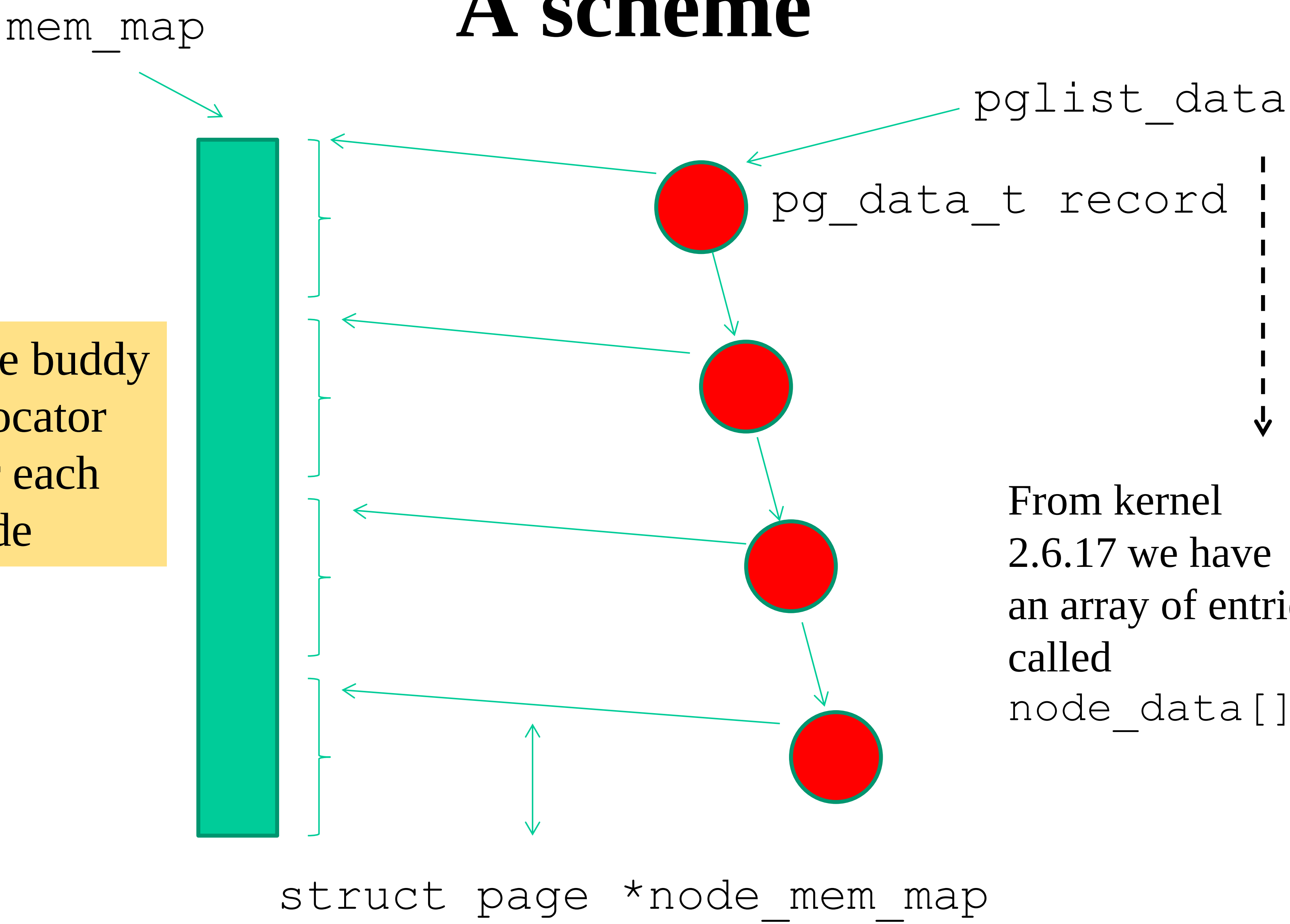
A scheme (picture from: Understanding the Linux Virtual Memory Manager – Mel Gorman)



Jumping to more recent kernels for actually NUMA machines – 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, as already seen called `contig_page_data` is used

A scheme



One buddy allocator per each node

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

Summarizing (still for 2.4 kernel)

- Architecture setup occurs via `setup_arch()` which will give rise to
  - a Core Map with all reseved frames
  - Free lists that looks like if no frame is available (in fact they are all reserved at this stage)
- Most of the work is done by `free_area_init()`
- This relies on bitmaps allocation services based on `alloc_bootmem()`

Releasing boot used pages to the free lists

```
static unsigned long __init
free_all_bootmem_core(pg_data_t *pgdat)
{
      ...............
      for (i = 0; i < idx; i++, page++) {
            if (!test_bit(i, bdata->node_bootmem_map)) {
`
      // il frame non deve restare riservato
                  count++;
                  ClearPageReserved(page);
                  set_page_count(page, 1);
                  __free_page(page);
            }
      }
      total += count;
      ...............
      return total;
}
```

Allocation contexts (more generally, kernel level execution contexts)

- Process context
  - Allocation is caused by a system call
    - 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 gives rise to kernel corruption

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

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)

Mempolicy details

- Generally speaking, mempolicies 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 mempolicies 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 a 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 levels paging, PGDs, PMDs and PTEs have two sets of functions each for the allocation and freeing of page tables.
- The allocation functions are `pgd_alloc()`, `pmd_alloc()` and `pte_alloc()`, respectively the free functions are, predictably enough, called `pgd_free()`, `pmd_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()
```

Quicklist API

```
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;
}
```

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

- The prototypes are in `#include <linux/malloc.h>`
- The main APIs are
  - `void *kmalloc(size_t size, int flags)`
    allocation of contiguous memory **<u>of a given size</u>** - it returns the virtual address
  - `void kfree(void *obj)`
    frees memory allocated via `kmalloc()`

- Main features:
  - Cache aligned delivery of memory chunks (performance optimal access of related data within the same chunk)
  - Fast allocation/deallocation support

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

What about large size allocations

- Classically employed while adding large size data structures to the kernel in a stable way
- This is the case when, e.g., mounting external modules
- The main APIs are:
  - `void * vmalloc(unsigned long size);`
    allocates memory of a given size, which can be non-contiguous, **and returns the virtual address** (the corresponding frames are anyhow reserved)
  - `void vfree(void * addr)`
    frees the above mentioned memory

Be careful of performance effects due to global TLB invalidations!!

Logical/Physical address translation for kernel directly mapped memory (not `vmalloc` one)

We can exploit the 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

kmalloc vs vmalloc: an overall 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)