

Adressräume und Speicherverwaltung in Linux

PD Dr. Reinhard Bündgen
buendgen@de.ibm.com

Virtueller Speicher und Adressräume

Reale und Virtuelle Adressräume

- Realer Adressraum
 - Menge der im Hauptspeicher physisch vorhandenen Adressen
- Virtueller Adressraum
 - Menge der in einem Programm logisch erreichbaren Adressen (Code, Stack, Heap, ...)
 - beginnt bei Adresse 0, je nach Architektur, 2^{16} , 2^{24} , 2^{31} , $2^{31} + 2^{30}$, 2^{48} Bytes groß (i. A. > realer AR)
 - oft nur lückenhaft genutzt
- Seite
 - zusammenhängender Speicherbereich der Größe 2^k , beginnend auf Adresse $n \cdot 2^k$
 - Adressräume werden in Seiten gleicher Größe eingeteilt
 - Typische Seitengrößen: 4 kB, 8 kB
 - riesige Seiten (huge pages): 1MB und größer

Seiten

- Seite
 - zusammenhängender Speicherbereich der Größe 2^k Bytes, beginnend auf Adresse $j \cdot 2^k$
 - Adressräume werden in Seiten gleicher Größe eingeteilt
 - Typische Seitengrößen: 4 kB, 8 kB

Adressraum der Größe 2^n mit Seiten der Größe 2^k :

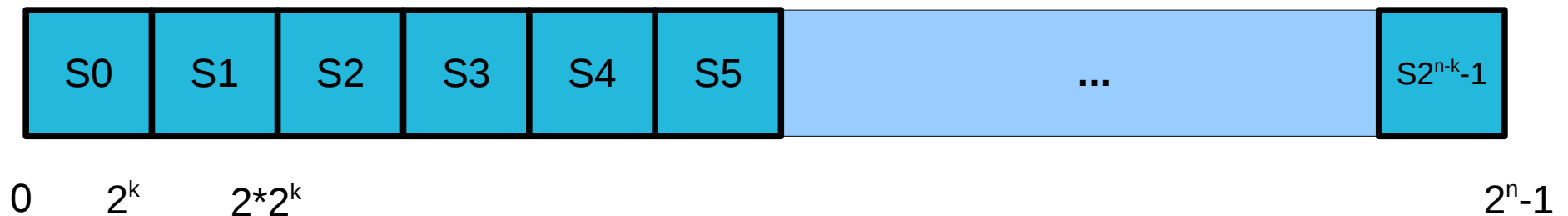
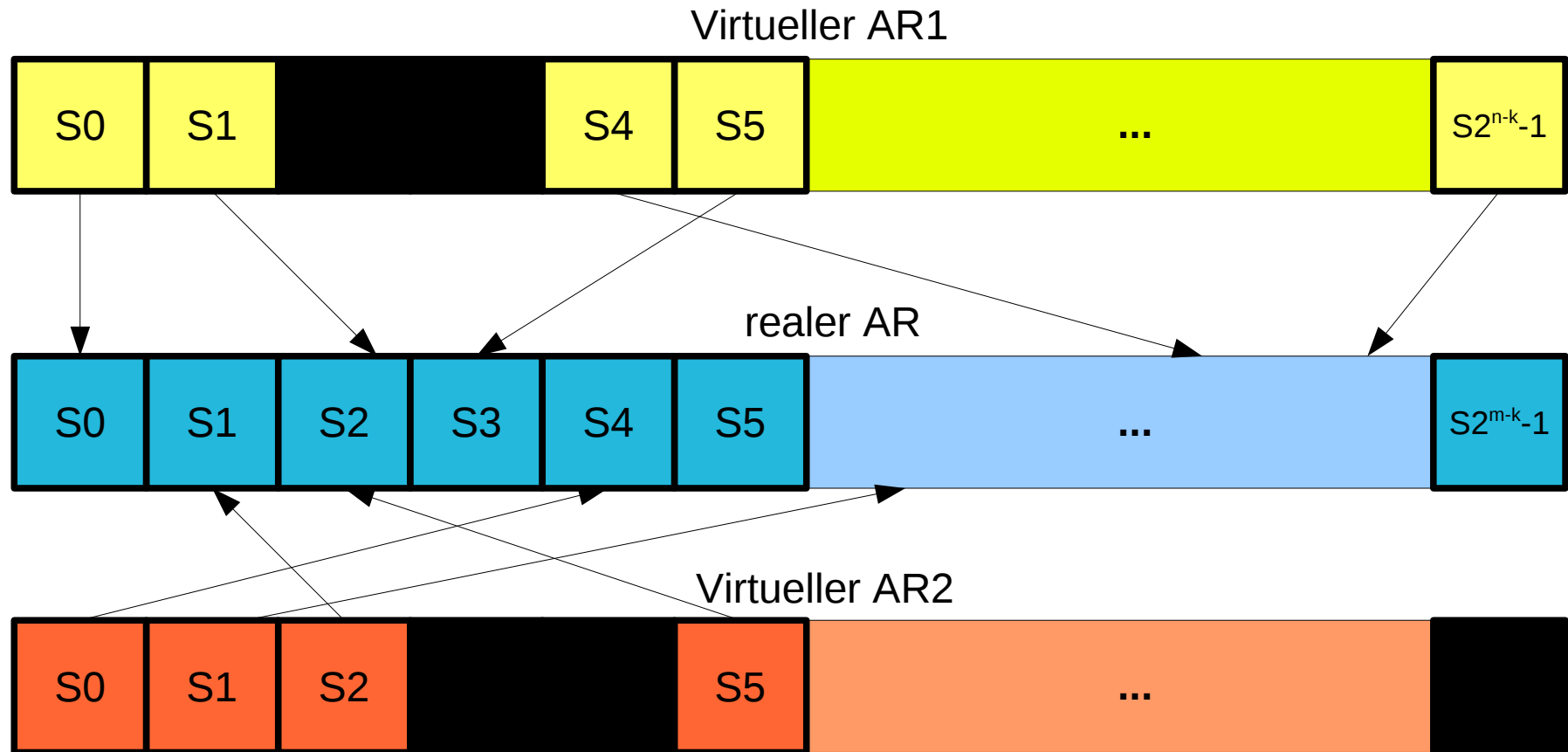


Abbildung von Virtuellen AR auf realen AR



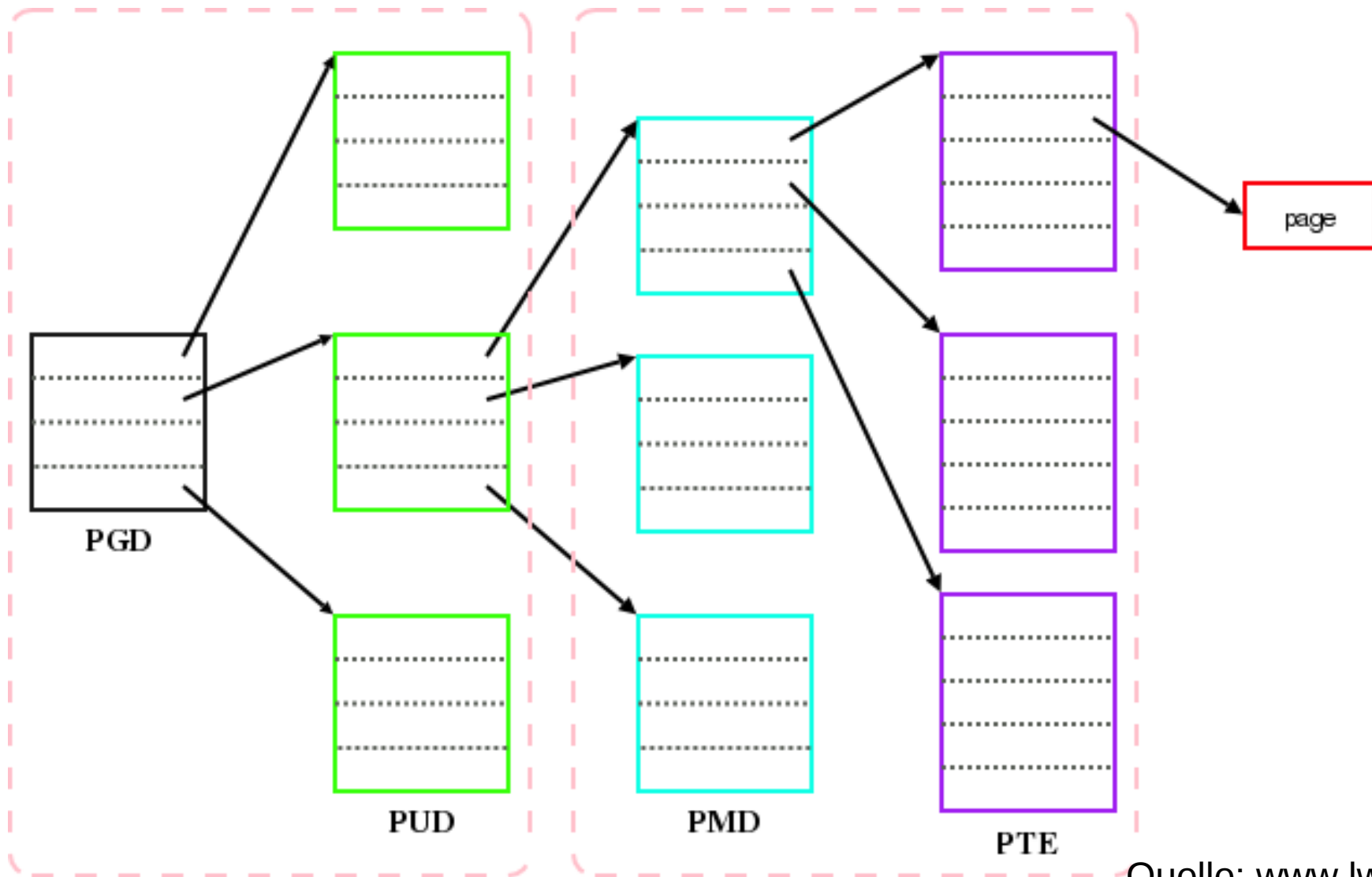
Seitenabbildungsfunktion

- Funktion, die *zum Zeitpunkt t* adressierbare Seiten des virt. AR *einer Task T* auf Seiten des realen AR abbildet
- i. A. ist zu einem beliebig aber festen Zeitpunkt t nicht der ganze virt. AR adressierbar!
- Wenn 2 Seiten zweier virt. AR auf die gleiche reale Adresse abgebildet werden, dann
 - gehört die Seite zum „shared memory“ beider zu den AR gehörigen Tasks oder
 - die Seiten sind „read only“

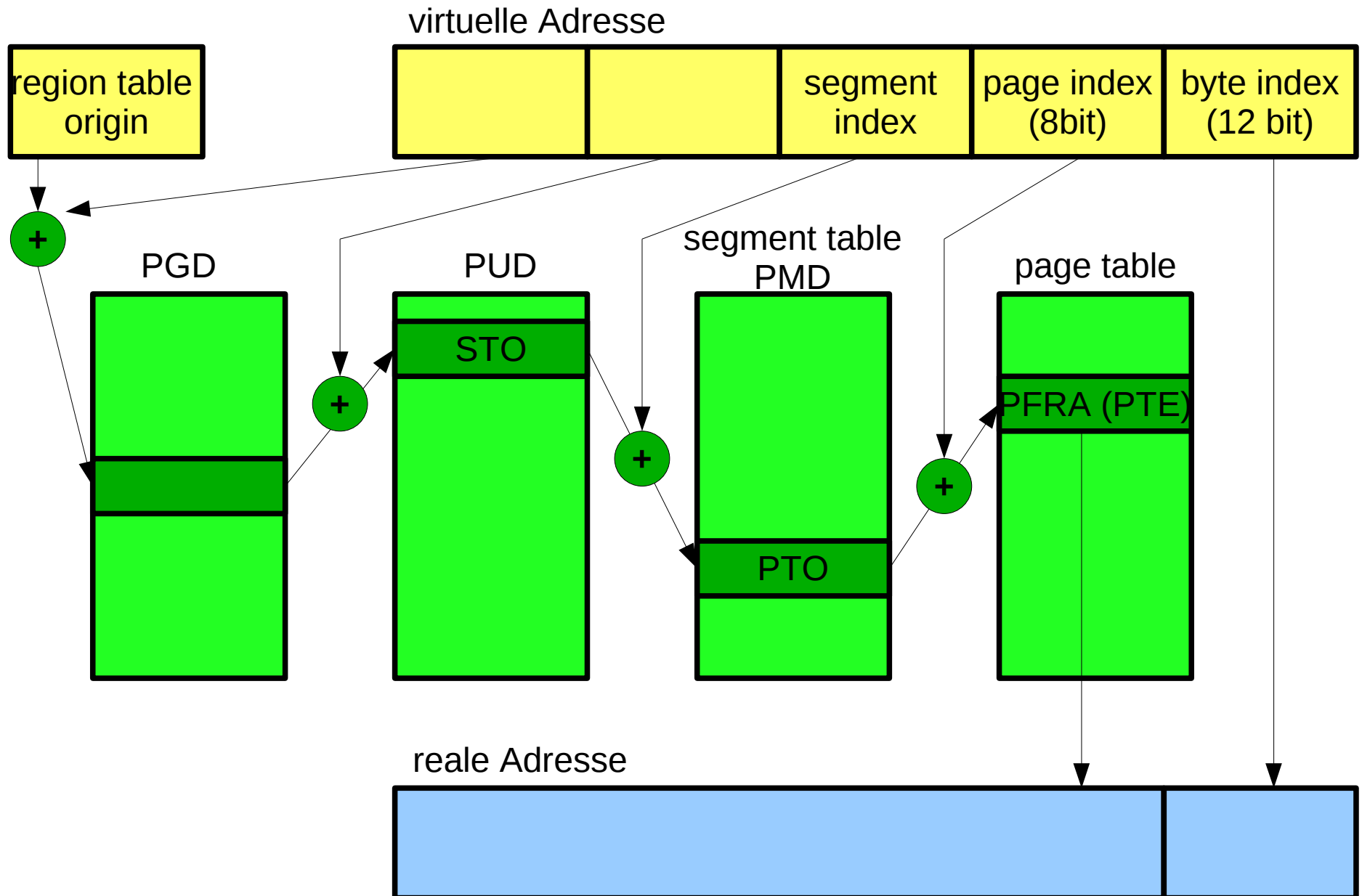
Realisierung der Seitenabbildungsfunktion

- mehrstufige hierarchische Seitentabellen:
 - unterste Hierarchie verweist auf Seiten
 - obere Hierarchien verweisen auf auf Tabellen
niederer Hierarchie
- Einträge in jeder Hierarchie dürfen leer sein, wenn keine dazugehörige Seite adressierbar ist.

Linux Seitentabellen: 4 Niveaus



Adressübersetzung



HW Unterstützung für virtuelle Adressierung

- mehrstufige Seitentabellen
 - z.B. i386: 2-stufig,..., s390x 5-stufig
 - caching der meist benutzten Adressen: TLB
- spezielle Register, die auf Tophierarchie der Seitentabellen zeigen
- page fault exceptions
 - Unterbrechungsmechanismus
 - ausgelöst, wenn PTE Seite als nicht vorhanden markiert

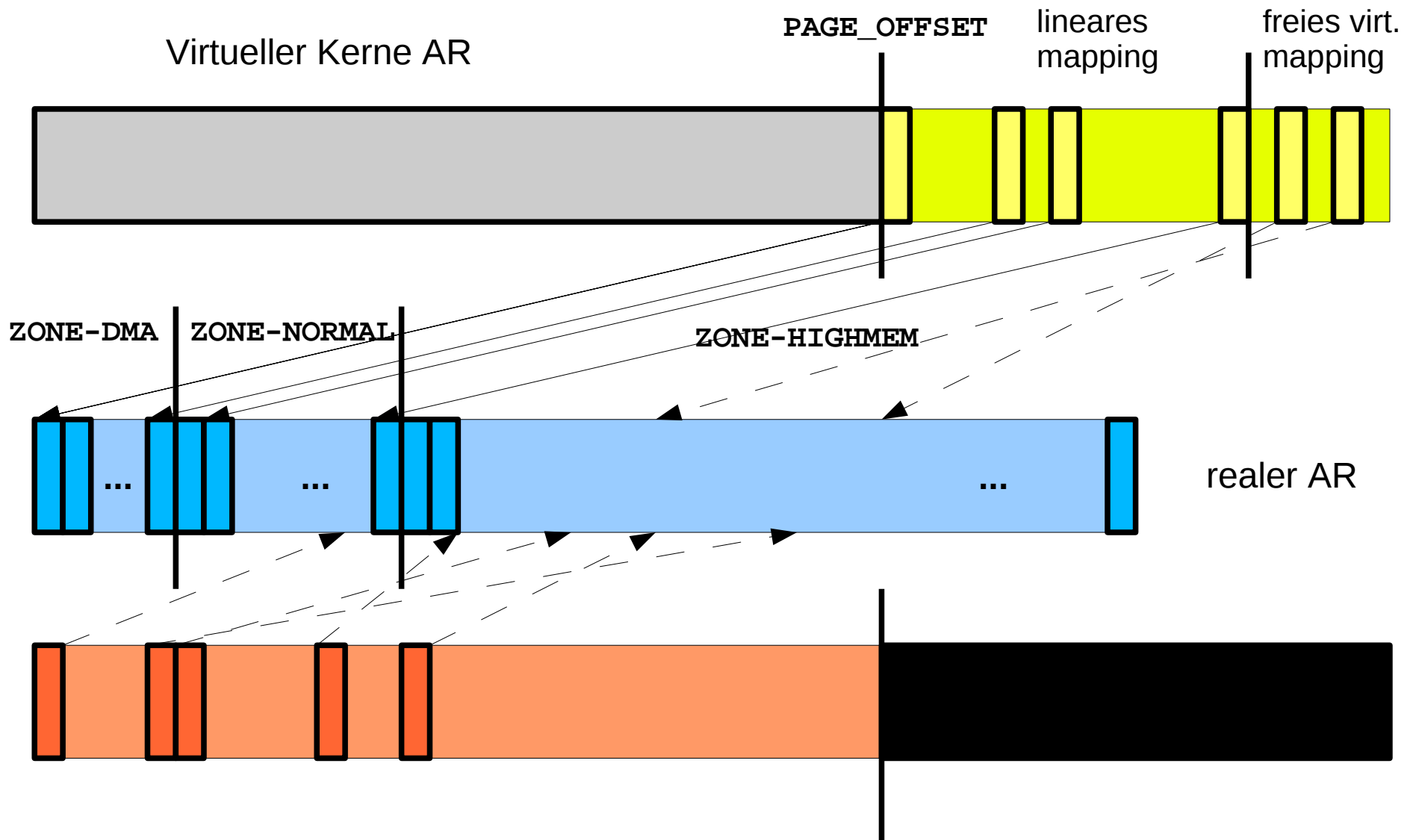
Vorteile der Virtuellen Adressierung

- gleichartige lineare Adressräume für alle Programme
- Speicherschutz:
 - verschiedene Adressräume referenzieren i. A. disjunkte Mengen realer Seiten
- es kann mehr virtueller Speicherplatz genutzt werden als real verfügbar ist
 - Lücken in virtuellem AR müssen nicht real vorhanden sein
 - on demand paging:
 - Seiten können von einer Platte eingelesen werden
 - Seiten können auf eine Platte ausgelagert werden

Linux Kernadressraum

- Boot:
 - temporäre Seitentabelle
 - umfasst Kerncode und Kerndatensegmente, Seitentabellen + 128KB für dynamischen Speicher
- Finale Kern Seitentabelle
 - erster Seitenbereich
 - lineares mapping
 - virtuelle Adr = reale Adr + `PAGE_OFFSET`
 - restlicher Speicherbereich
 - freies virtuelles mapping
 - benötigt z.B. für `vmalloc()`

Virtueller Kernel AR für i386



Linux Speicherverwaltung

Repräsentation realer Seiten

- jede reale Seite wird durch ein Objekt vom Typ `struct page` beschrieben
- siehe `<linux/mm_types.h>`
 - `_count`: Referenzzähler,
 - `_mapcount`: Zähler der referenzierenden page tables
 - `lru`: LRU Datenstruktur
 - `flags`: siehe `<linux/page-flags.h>`
 - `PG_dirty`, `PG_locked`, `PG_slab`,
`PG_mappedtodisk`, ...
- verbraucht ca 1% des Speichers
- wird hauptsächlich vom Swapper gebraucht

Speicherzonen

- ZONE_DMA
 - DMA fähige Seiten
 - Intel: > 16MB, System z: < 2GB
- ZONE_NORMAL
 - normal adressierbare Seiten
 - Intel 16 MB – 896 MB
- ZONE_HIGHMEM
 - dynamisch gemappte Seiten
 - Intel: > 896 MB

Speicherallokation im Kern

- alloziere ganze Seiten
 - `alloc_pages()`, ...
- alloziere Speicher für bestimmte Datenstrukturen
 - Slab Allokator
 - neu: Slub Allokator
- alloziere Speicher byteweise
 - phys. zusammenhängend: `kmalloc()`
 - virtuell zusammenhängend: `vmalloc()`

Allokation von Seiten

- `struct page *`
`alloc_pages(`
`unsigned int gfp_mask,`
`unsigned int order)`
- `unsigned long`
`__get_free_pages(`
`unsigned int gfp_mask,`
`unsigned int order)`
- `void __free_pages(`
`struct page * page,`
`unsigned int order)`
- `void free_pages(`
`unsigned long addr,`
`unsigned int order)`
- `get_zeroed_page()`
- `<linux/gfp.h>:`
- `__GFP_DMA`: Seiten aus DMA Zone
- `__GFP_HIGHMEM`: Seiten aus highmem Zone
- `__GFP_WAIT`: darf schlafen
- `__GFP_HIGH`: greife ggf auf Notfallreserven zu, (darf nicht schlafen)
- `__GFP_IO`: darf E/A auslösen
- `__GFP_FS`: darf Datei E/A auslösen
- `__GFP_COLD`: nutze Cold Page cache
- `__GFP_NOFAIL`: nie aufgeben
- `__GFP_NORETRY`: nur ein Versuch

Beispiel:

2 Seiten für Interrupt handler: `a = __get_free_pages(__GFP_HIGH, 1)`

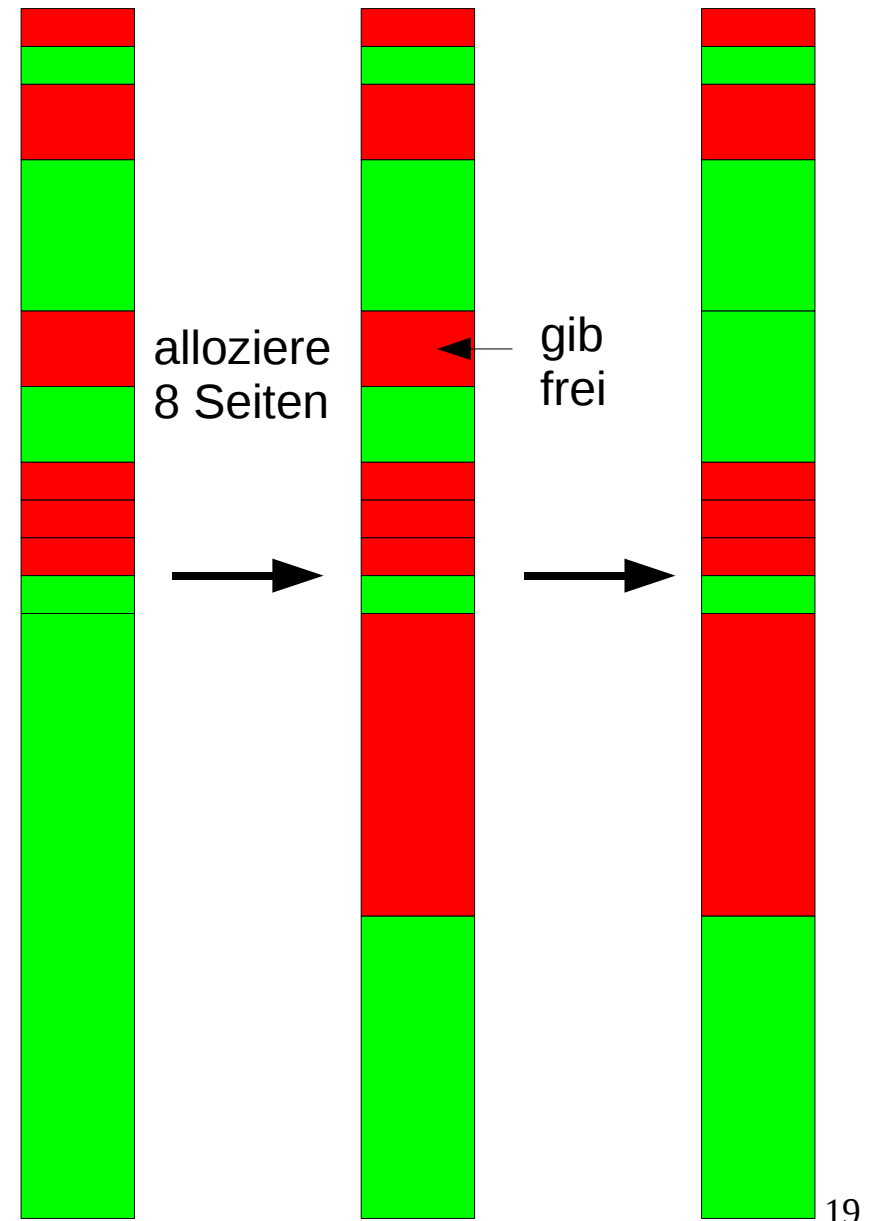
4 Seiten für DMA Treiber: `a = __get_free_pages(__GFP_DMA | GFP_KERNEL, 2)`

1 Seite für Nutzer: `p = alloc_free_pages(__GFP_WAIT | _GFP_IO | __GFP_FS, 0)`

Zonenbasierter Allokator

frei
benutzt

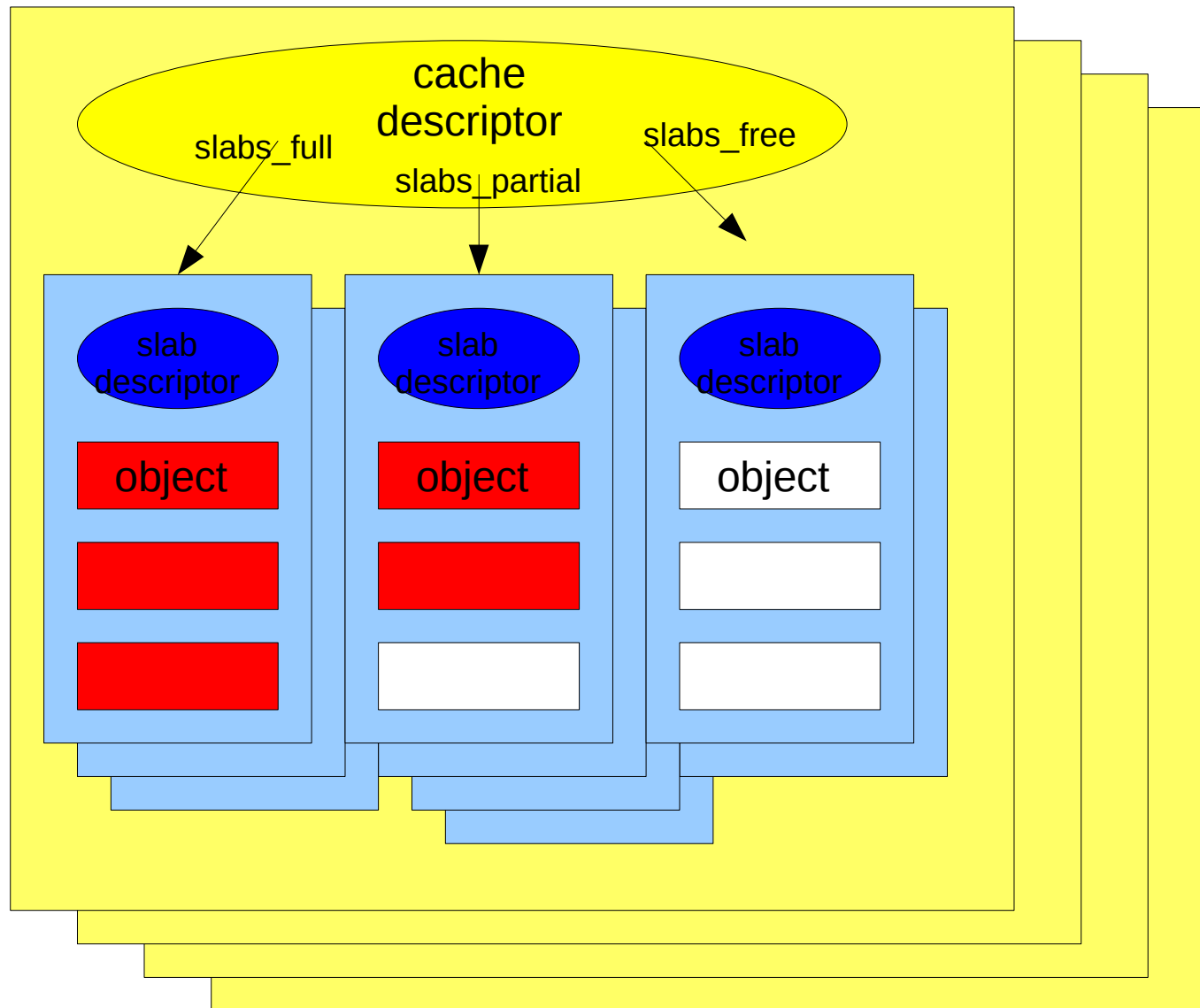
- Alloziert Seiten
- ein Suballokator pro Zone
- jeder Suballokator
 - betreibt ein Buddy System
 - hat je CPU einen Per-CPU frame cache
 - hot cache: in HW cache
 - cold cache
- Buddy System
 - alloziere 2^n Seiten
 - Freigabe: verschmelze Buddies



Der Slab Allocator

- Speicherverwaltung für Objekte die oft dynamisch erzeugt und gelöscht werden
- Vorbild Solaris 2.4
- ein Cache je Objekttyp
 - aufgeteilt in Slabs
 - jeder Slab besteht aus einer oder mehreren zusammenhängenden Seiten
 - jeder Slab kann maximal 1 oder mehrere Objekte des gleichen Typs enthalten
- Speicher gelöschter Objekte wird aufbewahrt für nächste Allokationen von Objekten dieses Typs
- Schwellenwerte für die Freigabe von Cachespeicher an Kernel
- `/proc/slabinfo`

Struktur des Slab-Allokators



SLAB Kernel API

- Erzeugung eines neuen Caches

```
kmem_cache_create(const char name, size_t size,  
size_t align, unsigned long flags, void  
(ctor)(void *, kmem_cache_t *, unsigned long),  
(dtor)(void *, kmem_cache_t *, unsigned long))
```

- Allocation neuer Objekte im Cache

```
kmem_cache_alloc(kmem_cache_t *cachep, int flags)  
kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

Allokation von Bytesequenzen

- Physikalisch zusammenhängend
 - `include/linux/slab.h` & `mm/slab.c`
 - `void * kmalloc(size_t size, int flags)`
 - `void kfree(const void *objp)`
 - Standard-Slabs der Größen $2^5 - 2^{17}$ Bytes
- virtuell zusammenhängend
 - `mm/vmalloc.c`
 - `void * vmalloc(unsigned long size)`
 - `void vfree(void * addr)`

SLUB Allokator

- seit 2.6.22 experimentell im Kern
- seit 2.6.23 default Allokator
- gleiches Kernel API wie SLAB Allokator
- Grund:
 - geringerer Overhead für Metadaten
 - besser Skalierbarkeit für viele CPUs
- Unterschiede zu SLAB
 - Metadaten in SLAB: nur free list
 - free list Verankerung über struct page
 - volle SLABs nicht SLUB Datenstruktur verbunden
 - eine partieller SLAB per NUMA Knoten