Memory Management and Virtual Memory

Memory Management, Virtual Memory – Slide Plan

What is Memory Management?

From an operating systems point of view, memory management happens at multiple levels:

- Hardware-assisted – Memory management unit (MMU) assisted address space segmentation and paging
- Software-assisted – Operating Systems management of memory allocations and freeing of virtual memory (kernel and user space) that has underlying MMU-mappings to physical addresses
- Software-assisted (Swap) – Operating Systems management of swapping from disk and corresponding freeing up of physical pages in RAM

Address Spaces

Let us look at hardware-assisted memory management first:

At a basic level, address spaces can be categorised as:

- Kernel/interrupt-level
- Process/user-level

Further to this, there is address spaces separation across processes
In addition, there are Hypervisor-assisted MMU management and IO-assisted MMU configurations - we won't look at these right now

Address Spaces – Terminology Explained

- Logical address = compiler generated address
- Virtual address = address as per segmentation unit (sometimes referred to as linear address)
- Physical address = actual hardware address listed in the Page Table Entry (PTE) used to access the physical memory address via the memory controller
- The mapping in the MMU (combination of segmentation + paging) = Logical → Virtual (linear) → Physical (actual)

Address Spaces Side Effects Summary

- On PowerPC (RISC), the segment register based separation
- Segment register content specific to process, and hence part of process context
- Lightweight process – on systems with no swapping, everything is fairly light weight on RISC these days – why?
  - ✓ Physical caches (virtually indexed, physically tagged) – no cache flushes. Tagged TLB's – no flushing of TLBs
- Logical caches, physical caches

Address Spaces Side Effects Summary (contd.)

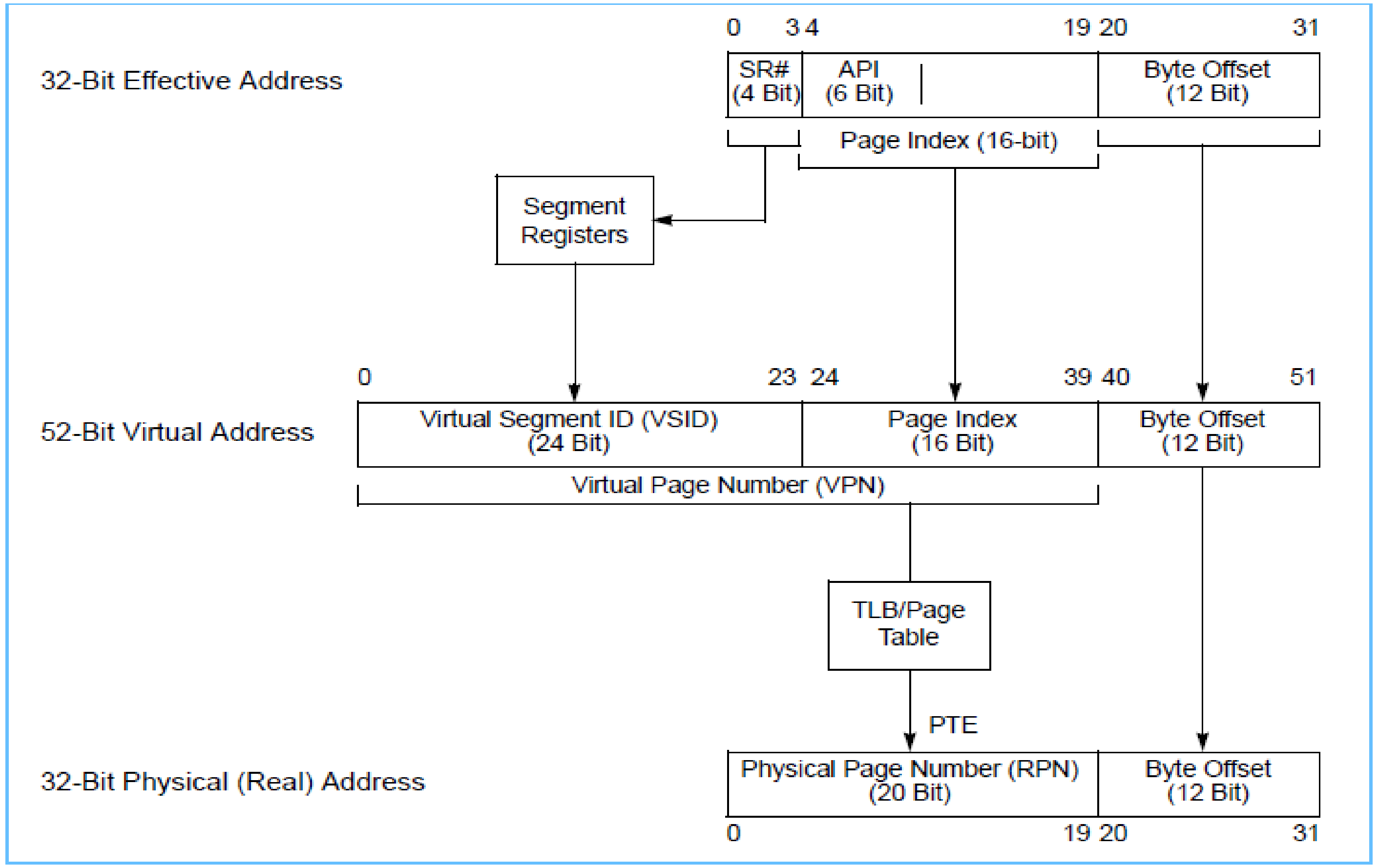
- On Intel x86, there is the overhead though of flushing TLB's on context switches as TLB's are not tagged
- Fork is a bit more time intensive than thread creation – why?
- All the why's will be answered in the following slides!

Kernel Memory Management Hardware – The Processor Interface

MMU internals – PPC and Intel x86 case studies

PowerPC MMU – Memory Mapping

Figure 7-15. Page Address Translation Overview—32-Bit Implementations



PowerPC MMU PTE

Figure 7-19 shows the format of the two words that comprise a PTE for 32-bit implementations.

Figure 7-19. Page Table Entry Format—32-Bit Implementations

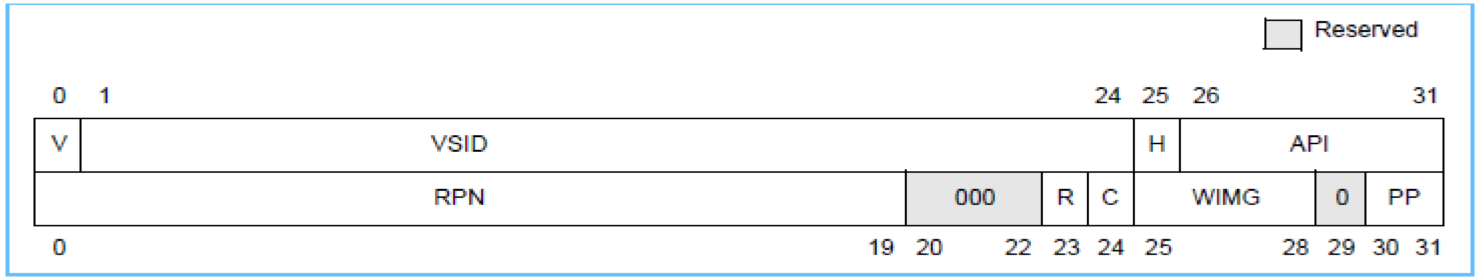
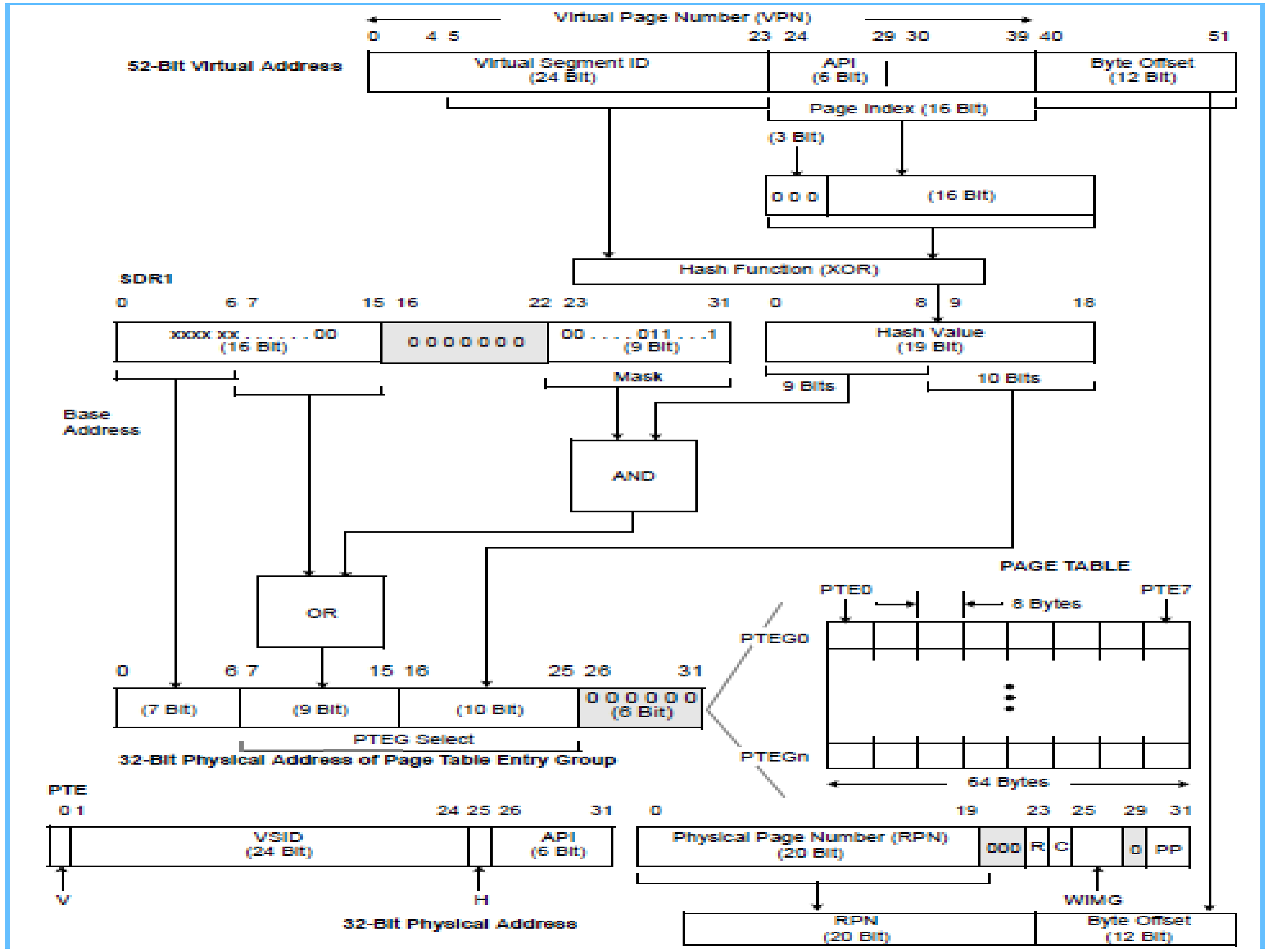


Table 7-16 lists the corresponding bit definitions for each word in a PTE as defined above.

Table 7-16. PTE Bit Definitions—32-Bit Implementations

Word	Bit	Name	Description
0	0	V	Entry valid (V = 1) or invalid (V = 0)
	1–24	VSID	Virtual segment ID
	25	H	Hash function identifier
	26–31	API	Abbreviated page index
1	0–19	RPN	Physical page number
	20–22	—	Reserved
	23	R	Referenced bit
	24	C	Changed bit
	25–28	WIMG	Memory/cache control bits
	29	—	Reserved
	30–31	PP	Page protection bits

PowerPC – PTE (contd.)



PowerPC – Cache, TLB lookup

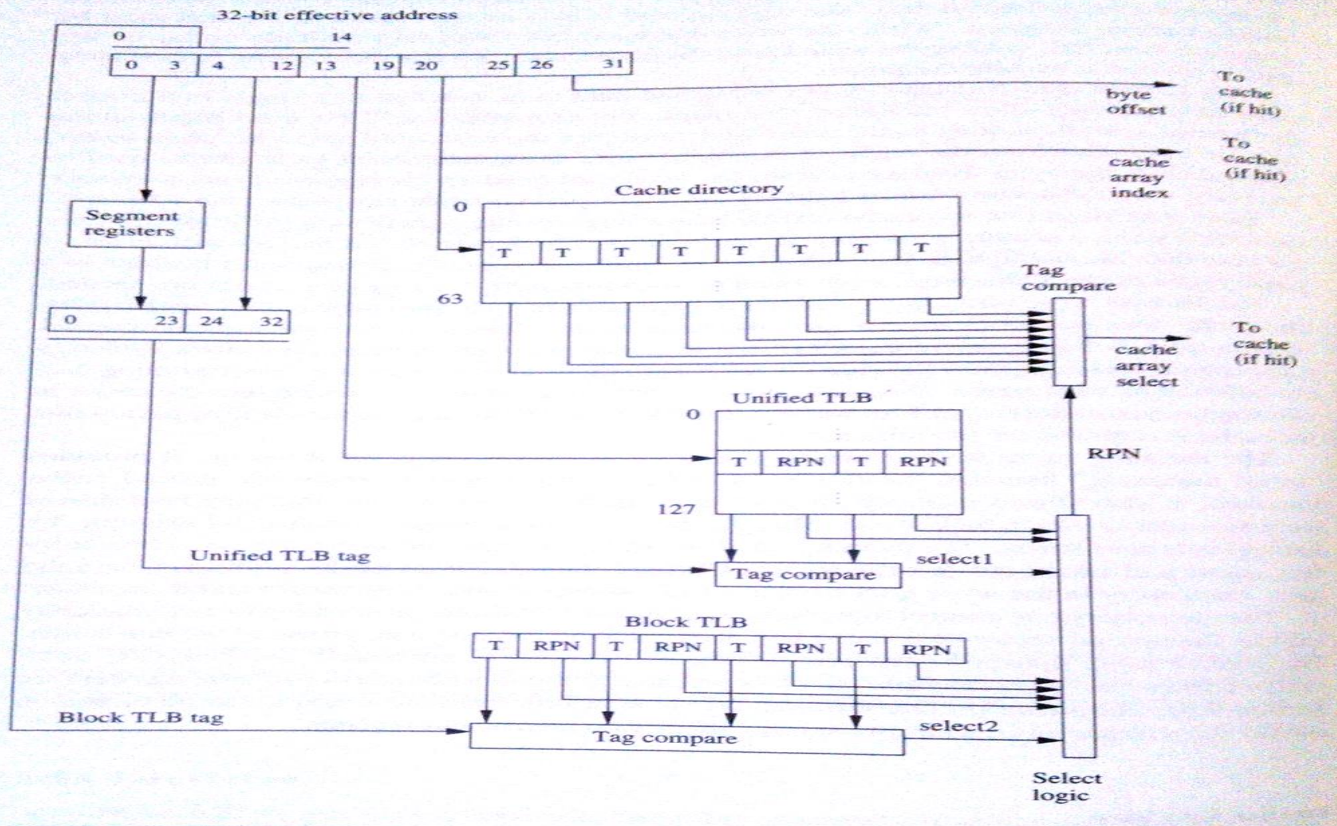


FIGURE 8.15 Cache lookup. T is tag. RPN is real page number.

Intel(x86) MMU, Segmentation Unit

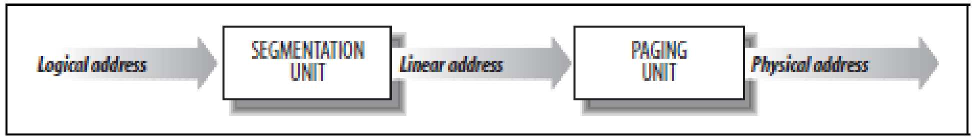


Figure 2-1. Logical address translation

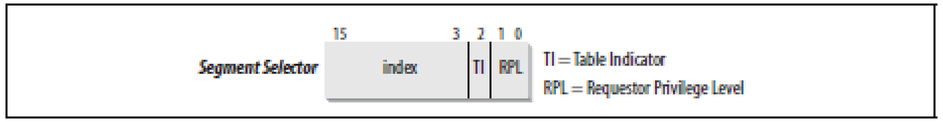


Figure 2-2. Segment Selector format

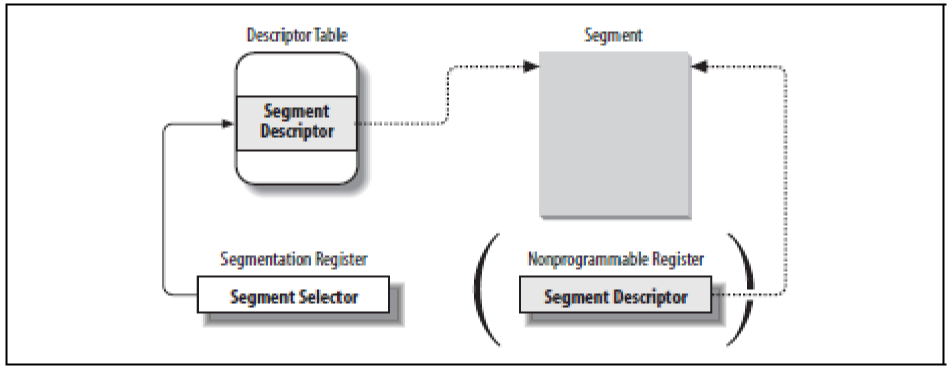


Figure 2-4. Segment Selector and Segment Descriptor

Table 2-2. Segment Selector fields

Field name	Description
`index`	Identifies the Segment Descriptor entry contained in the GDT or in the LDT (described further in the text following this table).
`TI`	*Table Indicator*: specifies whether the Segment Descriptor is included in the GDT (TI = 0) or in the LDT (TI = 1).
`RPL`	*Requestor Privilege Level*: specifies the Current Privilege Level of the CPU when the corresponding Segment Selector is loaded into the `cs` register; it also may be used to selectively weaken the processor privilege level when accessing data segments (see Intel documentation for details).

For example
CS: 32-bit compiler-generated logical address

CS contents are used to index into the Descriptor Table

Segmentation Unit (contd.)

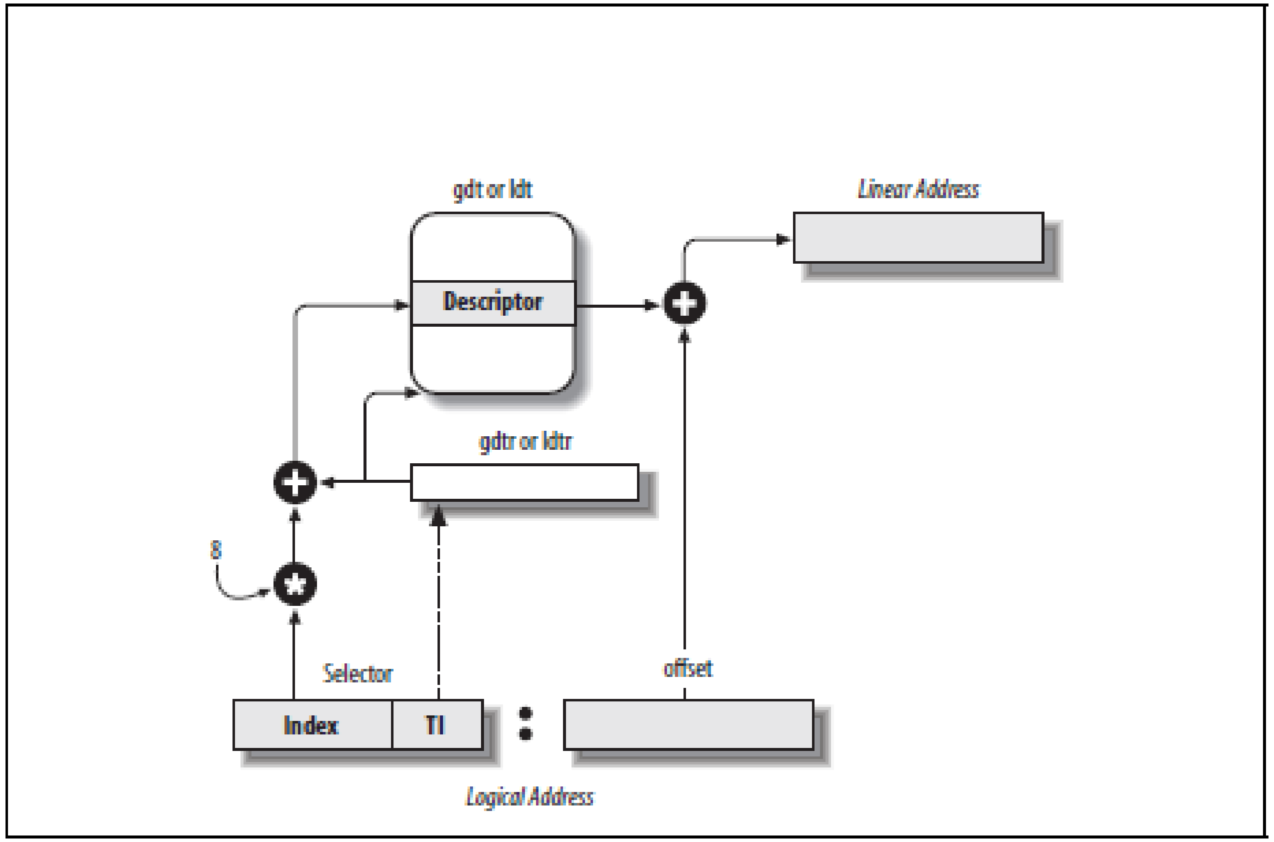
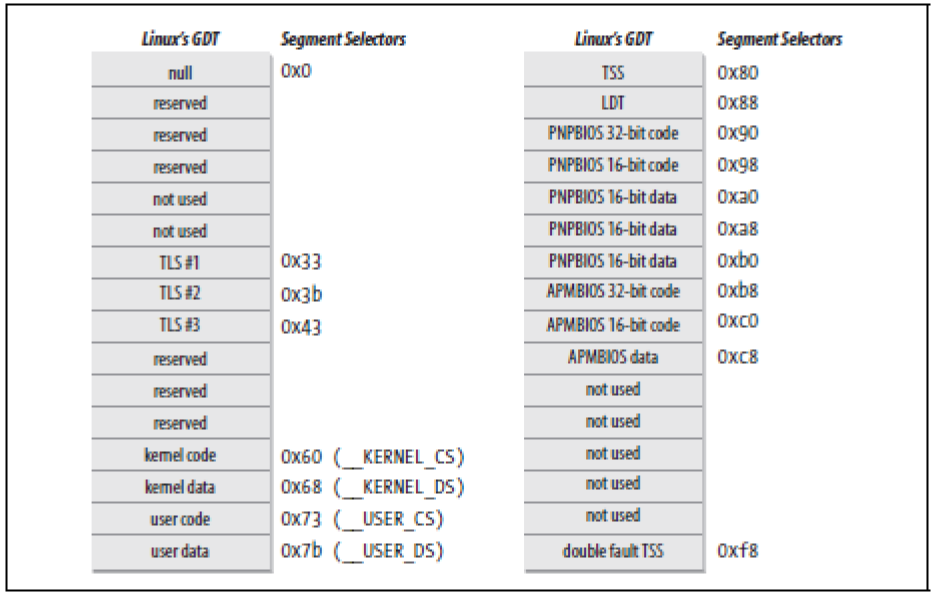


Figure 2-5. Translating a logical address

Segmentation Unit (contd.)



Linux's GDT	Segment Selectors	Linux's GDT	Segment Selectors
null	0x0	TSS	0x80
reserved		LDT	0x88
reserved		PNPBIOS 32-bit code	0x90
reserved		PNPBIOS 16-bit code	0x98
not used		PNPBIOS 16-bit data	0xa0
not used		PNPBIOS 16-bit data	0xa8
TLS #1	0x33	PNPBIOS 16-bit data	0xb0
TLS #2	0x3b	APMBIOS 32-bit code	0xb8
TLS #3	0x43	APMBIOS 16-bit code	0xc0
reserved		APMBIOS data	0xc8
reserved		not used	
reserved		not used	
kernel code	0x60 (__KERNEL_CS)	not used	
kernel data	0x68 (__KERNEL_DS)	not used	
user code	0x73 (__USER_CS)	not used	
user data	0x7b (__USER_DS)	double fault TSS	0xf8

Figure 2-6. The Global Descriptor Table

Take the case of __KERNEL_CS (arch/x86/include/asm/segment.h):

```
#define GDT_ENTRY_KERNEL_BASE 12
#define GDT_ENTRY_KERNEL_CS (GDT_ENTRY_KERNEL_BASE + 0)
#define __KERNEL_CS (GDT_ENTRY_KERNEL_CS * 8)
```

x86 MMU – Segmentation Unit

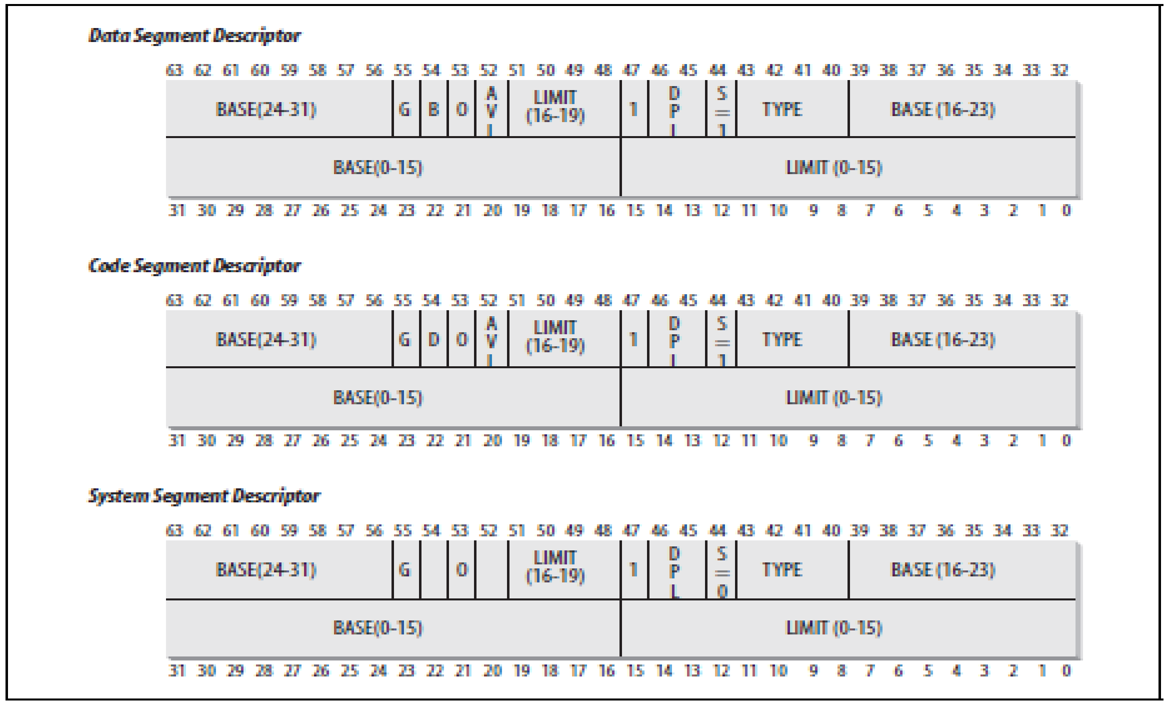


Figure 2-3. Segment Descriptor format

Table 2-3. Values of the Segment Descriptor fields for the four main Linux segments

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

x86 MMU – Segmentation Unit - Comments

- The x86 segmentation unit does not have the facility (that the PPC, Alpha etc. possess) to add an ASID or equivalent identifier to separate out process-specific address spaces
  - ✓ Hence, to keep things simple, the base and limit addresses of all descriptors are set to zero. As an effect, linear address = virtual address = offset that the compiler generates. (The address that ‘nm’ spews out = the linear address, and this is true for all executables, i.e. for all processes)

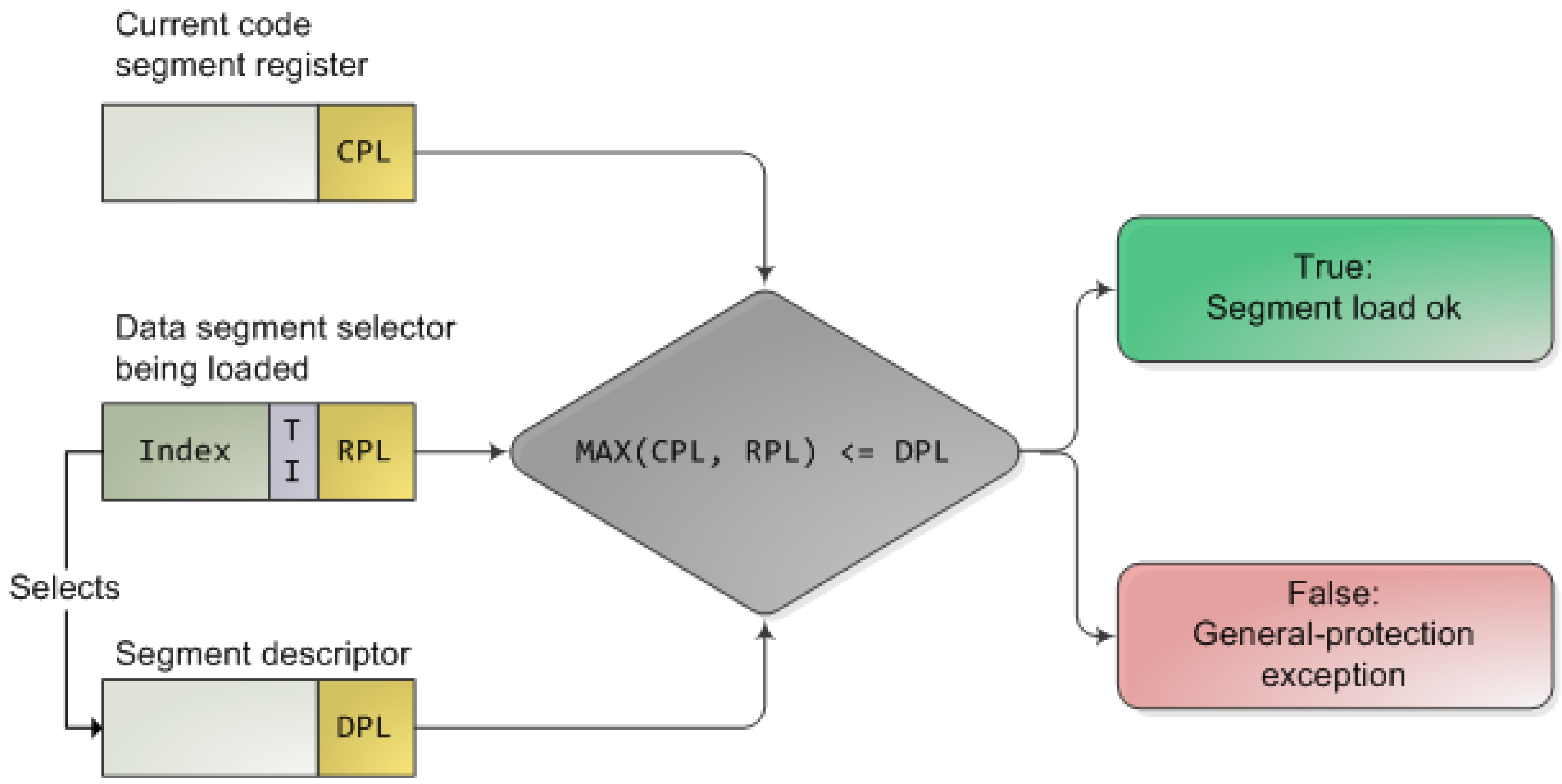
- On 64-bit Intel, the hardware base/limit check is taken out even

- Only the privilege levels vary across user/kernel space descriptors

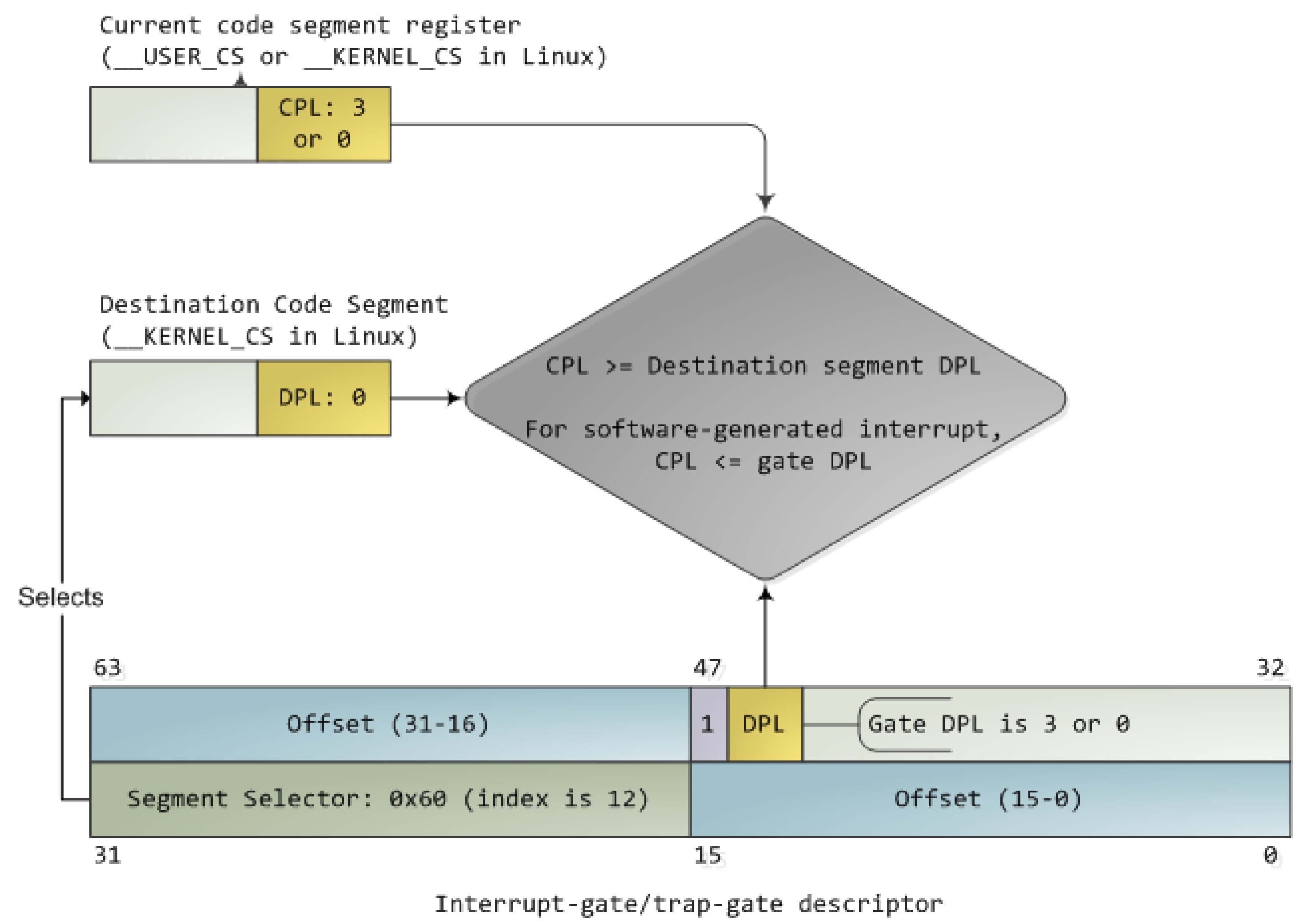
x86 MMU – Segmentation Unit – Comments (contd.)

- x86 MMU has a TLB that is not tagged by physical address/page number (again, unlike PPC, Alpha etc.)
  - ✓ Hence, every process switch will involve partial TLB flushes - portion of the TLB belonging to the switching-out process, not the kernel pages that are marked global
- CR3 always points to the base of the upper-most element of the page table hierarchy
- The partial flush is carried out by the loading of new process-specific CR3 register contents
- No CR3 change is done when switching between kernel threads

X86 Segmentation - Protection


Current code segment register
CPL
Data segment selector being loaded
Index
T I
RPL
Selects
Segment descriptor
DPL
MAX(CPL, RPL) <= DPL
True: Segment load ok
False: General-protection exception

X86 Segmentation - Protection



Current code segment register
(__USER_CS or __KERNEL_CS in Linux)
CPL: 3 or 0
Destination Code Segment
(__KERNEL_CS in Linux)
DPL: 0
CPL >= Destination segment DPL
For software-generated interrupt, CPL <= gate DPL
Selects
63
47
32
Offset (31-16)
1
DPL
Gate DPL is 3 or 0
Segment Selector: 0x60 (index is 12)
Offset (15-0)
31
15
0
Interrupt-gate/trap-gate descriptor

Intel (x86) MMU – Paging Unit

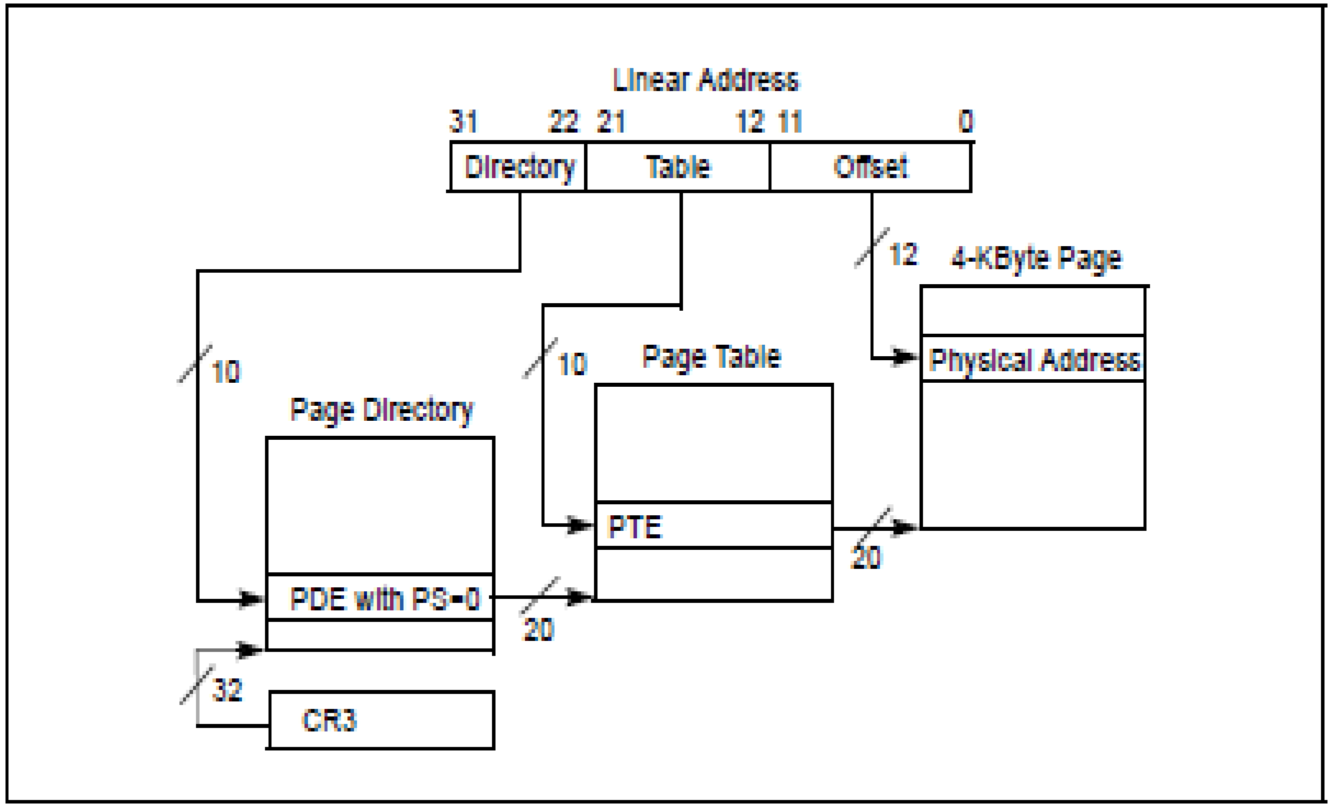


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Intel x86 – PAE – Paging Unit

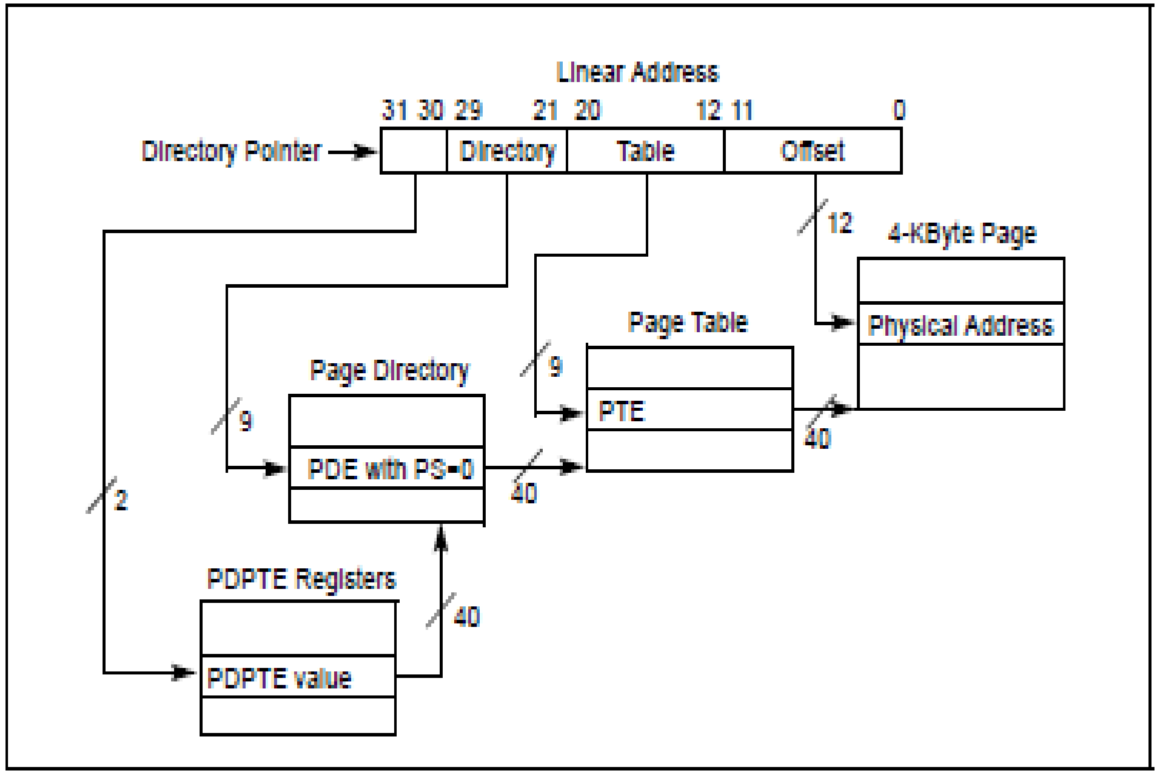
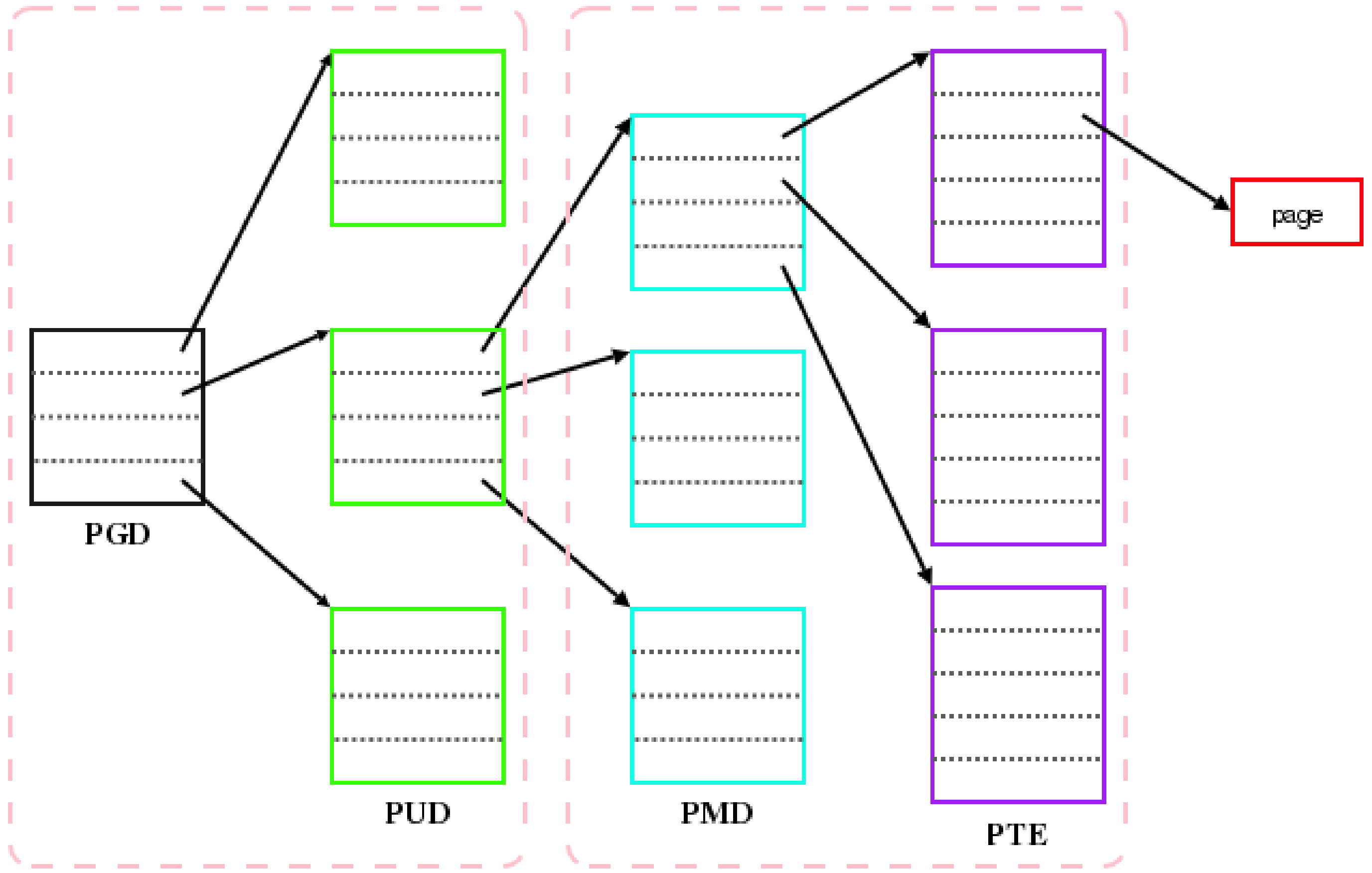


Figure 4-5. Linear-Address Translation to a 4-KByte Page using PAE Paging

Linux interworking with Intel MMU

- Multi-level (tree-based) page tables
- Hardware TLB miss-handling
- Linux uses a tree-based, hierarchical page table – this fits in with a lot of processor architectures including x86, and saves space for sparsely filled page tables (as compared to a single-level linear array)
- Names of the multiple levels in software are a bit confusing and don't correspond to the x86 hardware naming convention
- PGD = highest-level directory, PUD = page upper directory (only present for x86-64, collapses for 32-bit and PAE), PMD = page middle-level directory (collapses for 32-bit, but present for PAE), PTE = page table entry
- Collapse means pud_offset(pgd, address) would for instance return pgd because that's what the ifdef for PAE will compile to

Linux interworking with Intel MMU (contd.) – full 4-level page table (x86_64)



Copy-on-write

- Copy-on-write is a neat little trick implemented on Unix-based systems like Linux to save on space and time when a process does a fork.
- A fork is intended to clone a process (including the main thread)
- A fork is in a lot of places followed by an exec of a new binary however, and so fork-exec is a standard OS execution pattern to start the execution of a new application
- As this is the case, OS designers decide to use a processor MMU architecture setting to avoid potentially unnecessary copying of memory, in this case, the data sections, heap etc. of the original process

Copy-on-write(contd.)

- What is done on the fork is therefore: A copy of the original process's page tables are made for the newly forked process to use.
- A different CR3 (refer to PTE slides 20, 21) is used so that a copy is done of the PTE entries of the original process into the page tables of the new process
- And, the new mappings (same contents as the original process PTEs) in the forked process mark the physical pages as read-only so that when a write happens to any of the physical pages from this process's code, a trap happens and inside the trap handler, a new physical page is allocated and its PTE is updated accordingly

Dirty Pages

- A PTE can be marked as dirty.
- This facility is used when file writes happen to files that have contents loaded from disk into the page cache in RAM. When writes happen, the PTE's that correspond to pages in the page cache are marked dirty.
- These dirty pages are periodically, asynchronously (with respect to the file writes to the page cache) flushed to their original locations on disk.
- Only once this is done, can those page cache pages become potential spots to shrink the page cache, in low memory scenarios

Kernel memory allocation, user-space memory allocation – Slide plan

- Bird’s eye view – Slide 29
- Kernel memory allocation – Slides 30- 54
- User space memory allocation – Slides 55 - 61

Memory Allocation – Bird's Eye View

- Note: A lot of this content has a Linux bias, although the user-libraries and naming has a Linux flavour to it, similar ideas hold true for just about every OS.

- What you get:

  Malloc: Contiguous logical within user space, non-contiguous physical (usage: normal user space allocation)

  Kmalloc: Contiguous logical and physical (usage: kernel memory allocation, DMA areas)

  Vmalloc: Contiguous logical within kernel space, non-contiguous physical (kernel module space allocation)

- How you get it – how the user space library manages virtual memory, how kernel manages physical memory

- The additional complexity of logical→ physical address space mappings

Kernel memory allocation, page table updates – kernel interactions

Change of entry – fresh kernel space allocation

- Find free physical space – search lists of free blocks whose sizes are of order 2^(n-1) page size blocks, where n = 0 .. MAX_ORDER-1 (get_free_pages)
- Buddy, coalescing – if you can't find free blocks from within the appropriate order list (best fit) , go to the next best fit
- Split the next best fit and mark the split halves as buddies to reduce internal fragmentation
- Coalesce when freeing a buddy to reduce external fragmentation

Kernel memory allocation, page table updates – kernel interactions (contd.)

- Manipulating the page table – mark PTE as present (address mapping already set up by early kernel code, page table accessed using self-mapping trees)
- PPC – cache of the first-level of page table tree is the hash table
- No change to entry
  - **Keep a group of pre-allocated, content-initialized pages, use the slab allocator to point new data structures to pre-existing slab caches**
    - ✓ Performance factor of not having to allocate and initialize
    - ✓ Used generally for relatively smaller (less than KMALLOC_MAX_CACHE_SIZE) allocations to reduce internal fragmentation
- Change of entry – Swap – covered later

Buddy Allocator

- Basic data structure: Array of lists of different orders (2^n pages, n=0, 1, 2 ... 10)

- Algorithm:
  - Find the best fit, by using the order larger than or equal to block size requested – closest to the requested size
  - If best fit not found, go to the next available order, split it into two equal halves (or however many equals, depending on the order), mark the halves/equals as buddies to the chosen block and add them to the list of free blocks of the corresponding order
  - When freeing blocks, check for buddies, if free buddies found, coalesce
  - Splitting to reduce internal fragmentation, coalescing to reduce external fragmentation

The Slab Allocator

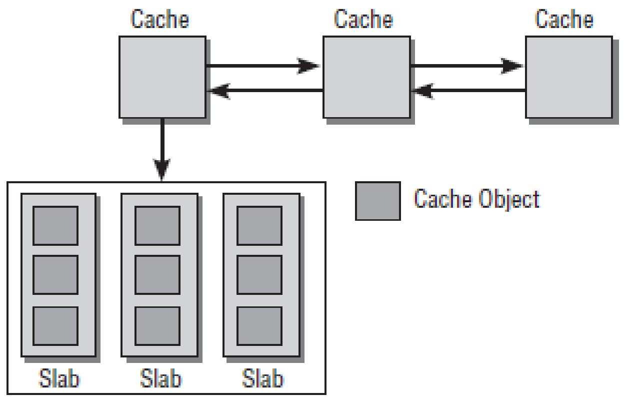


Figure 3-44: Components of the slab allocator.

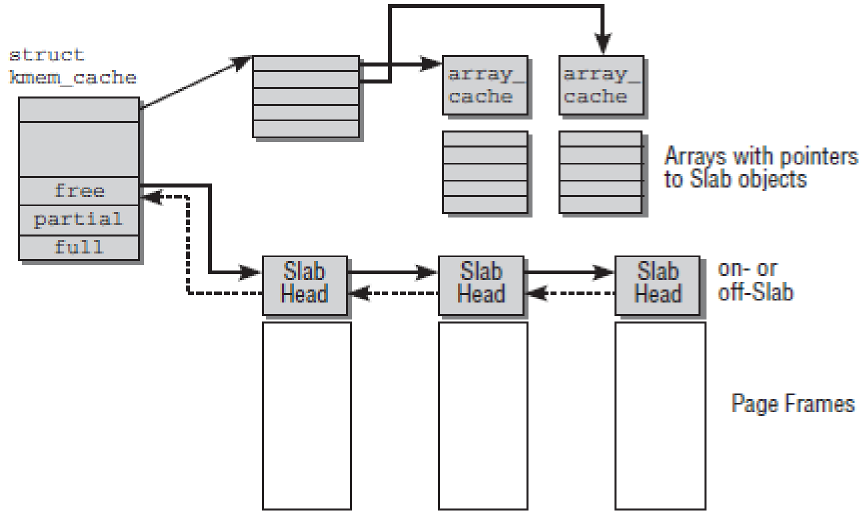


Figure 3-45: Fine structure of a slab cache.

The Slab Allocator (contd.)

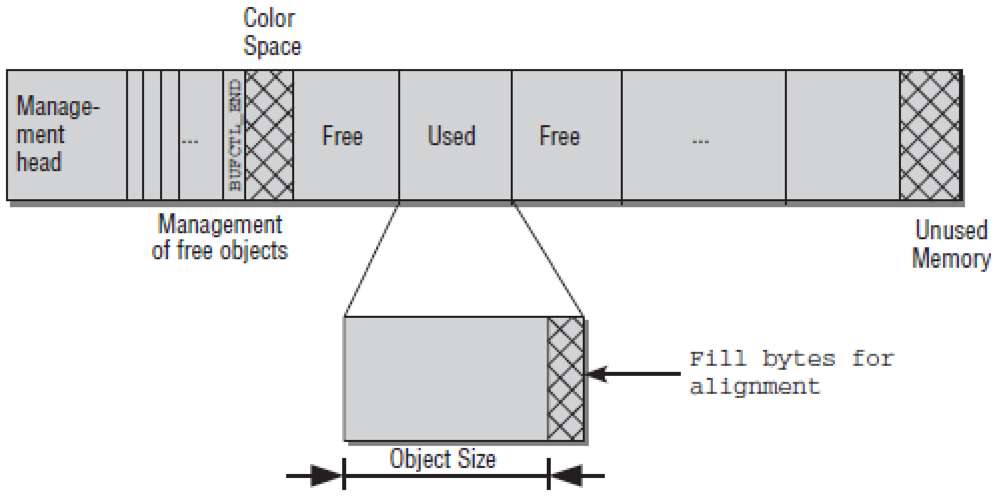


Figure 3-46: Fine structure of a slab.

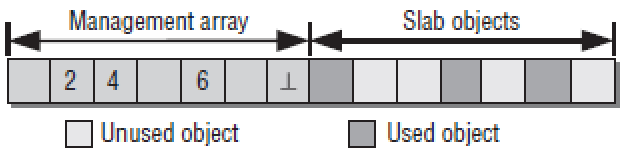
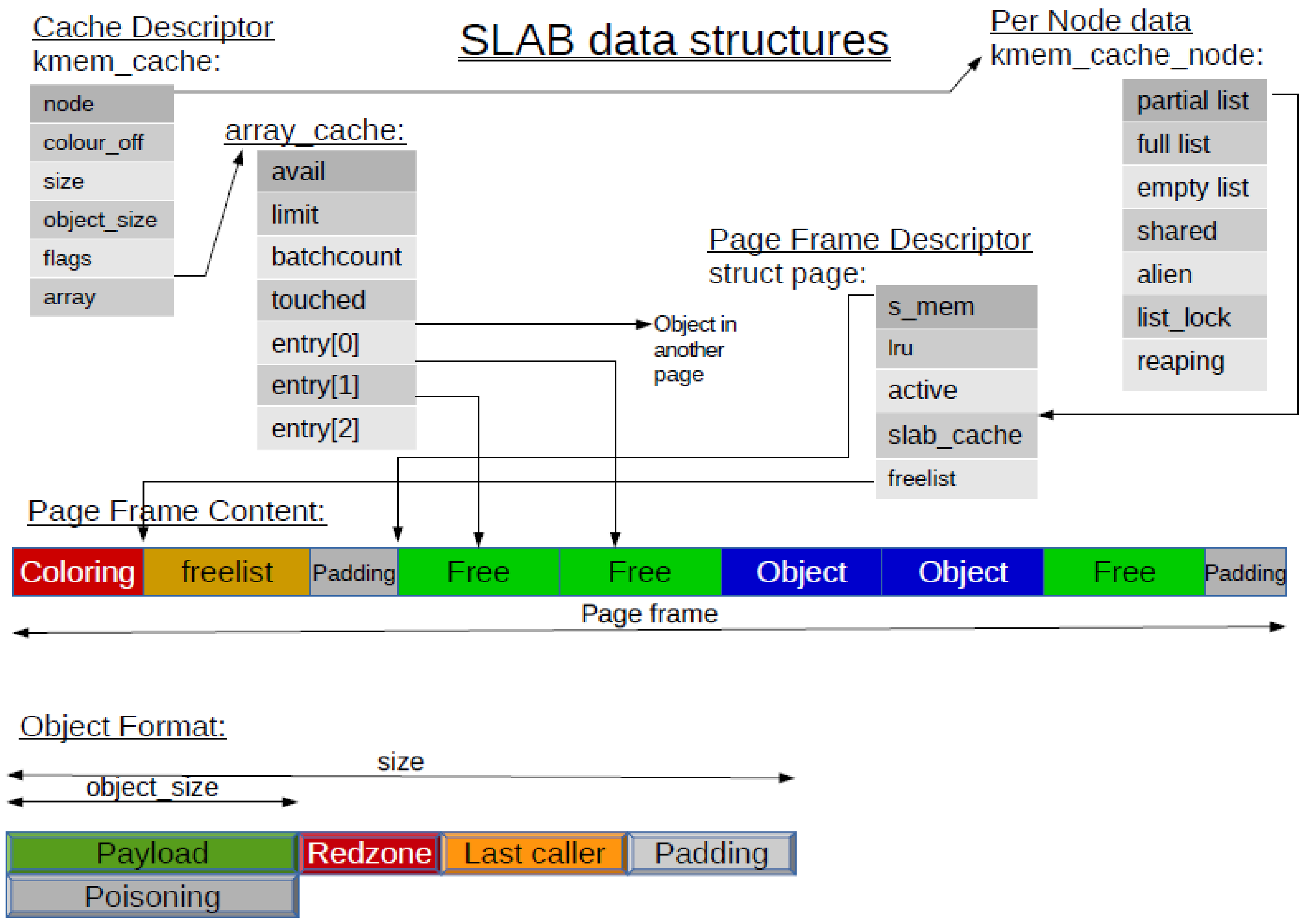


Figure 3-47: Management of the free objects in a slab.

The Slab Allocator (slightly evolved version) – more details



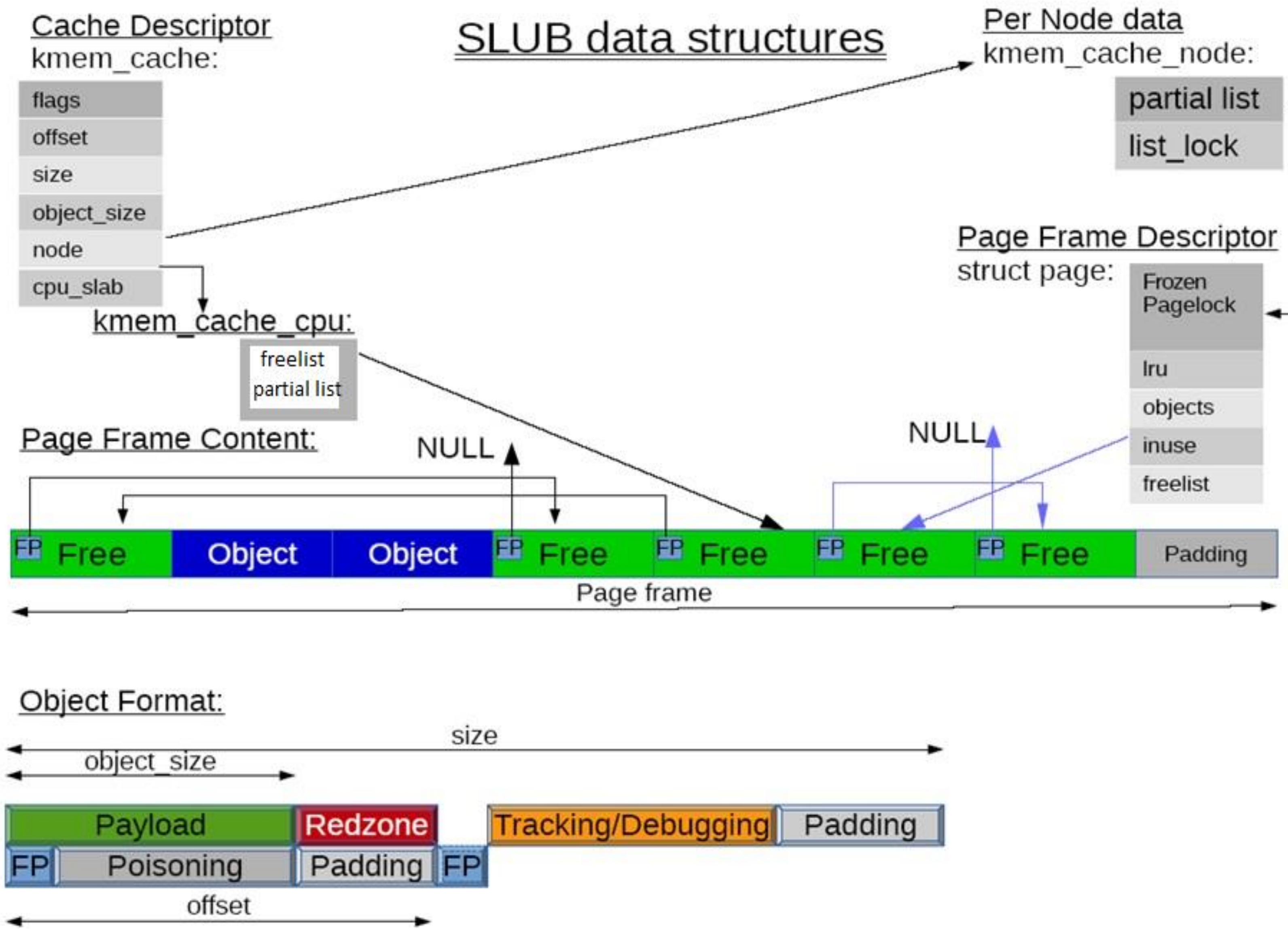
The Slab Allocator (contd.)

- Partial, free/empty, full slabs per node – hierarchy of slab caches
- LIFO-last-freed objects in slab stored in per-cpu array caches (potentially across pages)
- Free list (per-CPU, per-page), partial list (per-node)
  - ✓ The list of caches connects caches of all sizes. How do you get to the cache of a required size? Via a global pointer that is initialized when the specific cache is created
- Slab colouring – power of 2 polluting the cache, hence consecutive slabs are offset from page boundaries by different “colours”

The Slab Allocator (contd.)

- **Actual functioning:**
  - Size caches – kmalloc, kfree are mapped to appropriate size caches. Kmalloc locates per-cpu slab (if first allocation in slab, it gets the free list from the struct page's freelist - page from kmem_cache - and makes the per-cpu slab freelist point to this) from the size cache. If not found, it goes to the next slab in per-cpu partial list, and eventually to the node's partial list if needed
  - Kfree goes from kmem_cache to struct page and then to slab cache (per-CPU) and then to the specific partial list or if full, then the slab is got from the appropriate struct page fields and gets added to the per-CPU partial list
  - Similarly for kmem_cache_alloc, and kmem_cache_free
  - Note: Slab is effectively part of struct page in the slab allocator's evolved version

The Slub Allocator



The Slub Allocator (contd.)

- **Freelist management does not require space at the beginning of the page: it is now part of struct page and is direct pointer based - the first free object in the page is pointed to by the pointer in struct page, the rest of the list is made up of pointers within the free objects**
  - This makes cache-line alignment of objects easier

- **Merging of caches of same sizes**
  - This leads to lesser external fragmentation and better cache locality

The Slub Allocator (contd.)

Per-CPU lockless partial list with a per-node partial list

- Better locality without compromising on scalability
- First check per-CPU free list (that points to free objects from the first slab), then fallback to per-CPU partial list
- Fallback to per-node partial list, if per-cpu partial list is empty
- Fallback to across nodes, in increasing NUMA distance
- Fallback further to page allocator, if across-node search also turns up empty
- Migration of slabs from per-CPU to per-node (active→ unused, frozen/inuse=0)

The Slub Allocator (contd.)

No per-CPU array_cache entry array that points separately to the most recently freed objects from the slab, potentially across pages. Only one per-CPU free list

- **Inuse:** Once all struct pages that comprise a partial list have inuse fields of 0, the corresponding partial list can be freed and returned to the kernel
- **Frozen:** If a page is frozen, then no remote-to-the-node allocator can grab objects from this page

Kernel Memory Allocation - Review

- **Zone allocator (ZONE_DMA, ZONE_NORMAL , ZONE_HIGHMEM)**
  - ✓ ZONE_DMA – legacy bus requirement, addressability constraint
  - ✓ ZONE_NORMAL – where most kernel memory allocation happens
  - ✓ ZONE_HIGHMEM – user space memory allocation, plus physical space for kernel HIGHMEM mapping

Kernel Memory Allocation – Review (Contd.)

Physically contiguous virtual addresses (kmalloc)

Within zone:

- Generic standard repeated allocations of the same size get pushed into the slabs (like the malloc bins)

- Slab allocator cache (performance plus internal fragmentation avoidance) – used for standard types that the kernel normally allocates. For instance, there will be one slab cache for inodes, one for task_struct etc. Slabs point to objects of specific types (inodes, task_struct) or to blocks of specific sizes

- Kmalloc sits on top of the slab allocator

- Slab colouring is to decrease cache pollution

Kernel Memory Allocation – Review (Contd.)

- Fragmentation avoidance – buddy allocator – keeping track of contiguous free page frames for splitting/ coalescing. Splitting to reduce internal fragmentation, coalescing to reduce external fragmentation
- Buddy for larger allocations
- Malloc never fails, kmalloc will if there is not enough contiguous (physically) memory
- What do you need to specify when you call kmalloc – size plus a flag.
- Generally used flags are: GFP_ATOMIC, GFP_KERNEL

Kernel Memory Allocation – Review (Contd.)

- Pre-allocated page frames are used in case GFP_ATOMIC is specified in case of low memory because GFP_ATOMIC can't (is not designed to) sleep. Interrupt handlers cannot sleep and hence use GFP_ATOMIC
- GFP_KERNEL – can sleep while inactive pages are swapped out to disk or caches are shrunk to make room
- Performance - fast slab allocator (cache – per-CPU cache, avoiding spinlocks)
- Memory pools for specific uses in case of low memory, only to be used for specific purposes, like keeping some memory aside for slabs

Physically non-contiguous virtual addresses – vmalloc – generally used for kernel module space allocation

The Linux Virtual Address Space - Configuration

- User virtual address space – 0x0000000 (if 32-bit, 0x08048000, if 64-bit 0x0000000000400000) – 0xBFFFFFFF (3 GB)
- Low user virtual address space is used to map in libc libraries etc.
- Kernel virtual address space – 0xC0000000 (PAGE_OFFSET) – 0xFFFFFFFF (1 GB)
- Kernel virtual address space maps to 0x00100000 (1 MB mark from physical page 0 onward) – the first 1 MB is used for things like POST results
- The first 8 MB starting at virtual address 0xC0010000 (physical address 0x00100000) is where the kernel image is located

The Linux Virtual Address Space – Configuration (contd.)

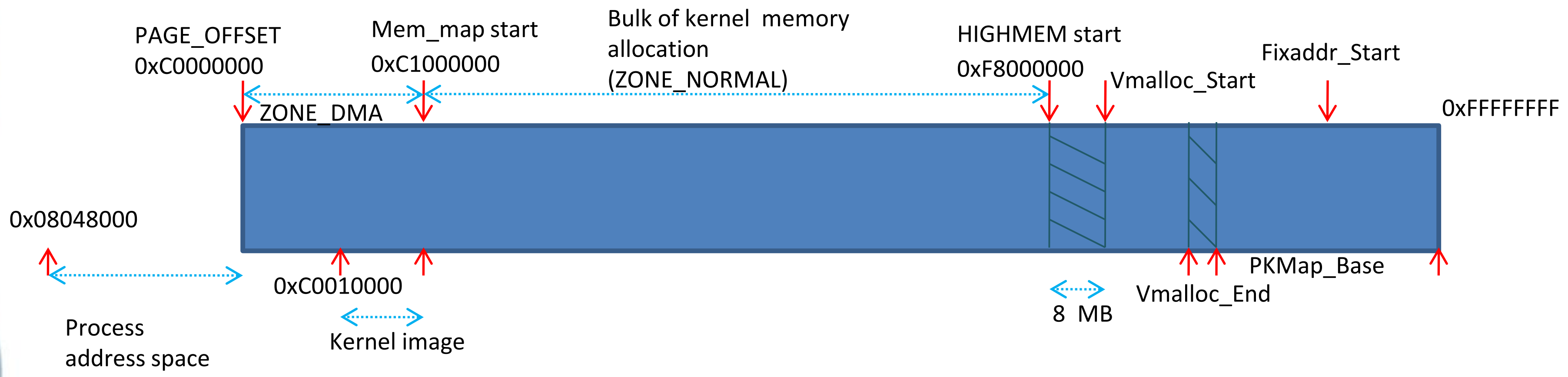
- The first 16 MB (including the above 8 MB) is allocated to ZONE_DMA
- Therefore, the first virtual address that is used for kernel purposes is 0xC0010000 + 16 MB = 0xC1000000 – which is where mem_map is located
- The lower 896 MB of this area after ZONE_DMA is used for permanent kernel mapping
- The higher 128 Mb of this area after ZONE_DMA is dedicated to temporary mappings and vmalloc

The Linux Virtual Address Space – Configuration (contd.)

- HighMem - temporarily mapped into kernel space in the upper 128 MB of the 1 GB kernel virtual memory) - use of HighMem when available kernel physical memory is 1 GB – the available virtual memory is restricted to less than 1 GB by vmalloc , mem_map and PTE requirements, hence you need some virtual space to temporarily map part of this 1 GB to parts of this HighMem-designated 128 MB. Mention PAE ?? (Without PAE no highmem??)

- Mem_map - array of struct page corresponding to every page – why we shall soon see

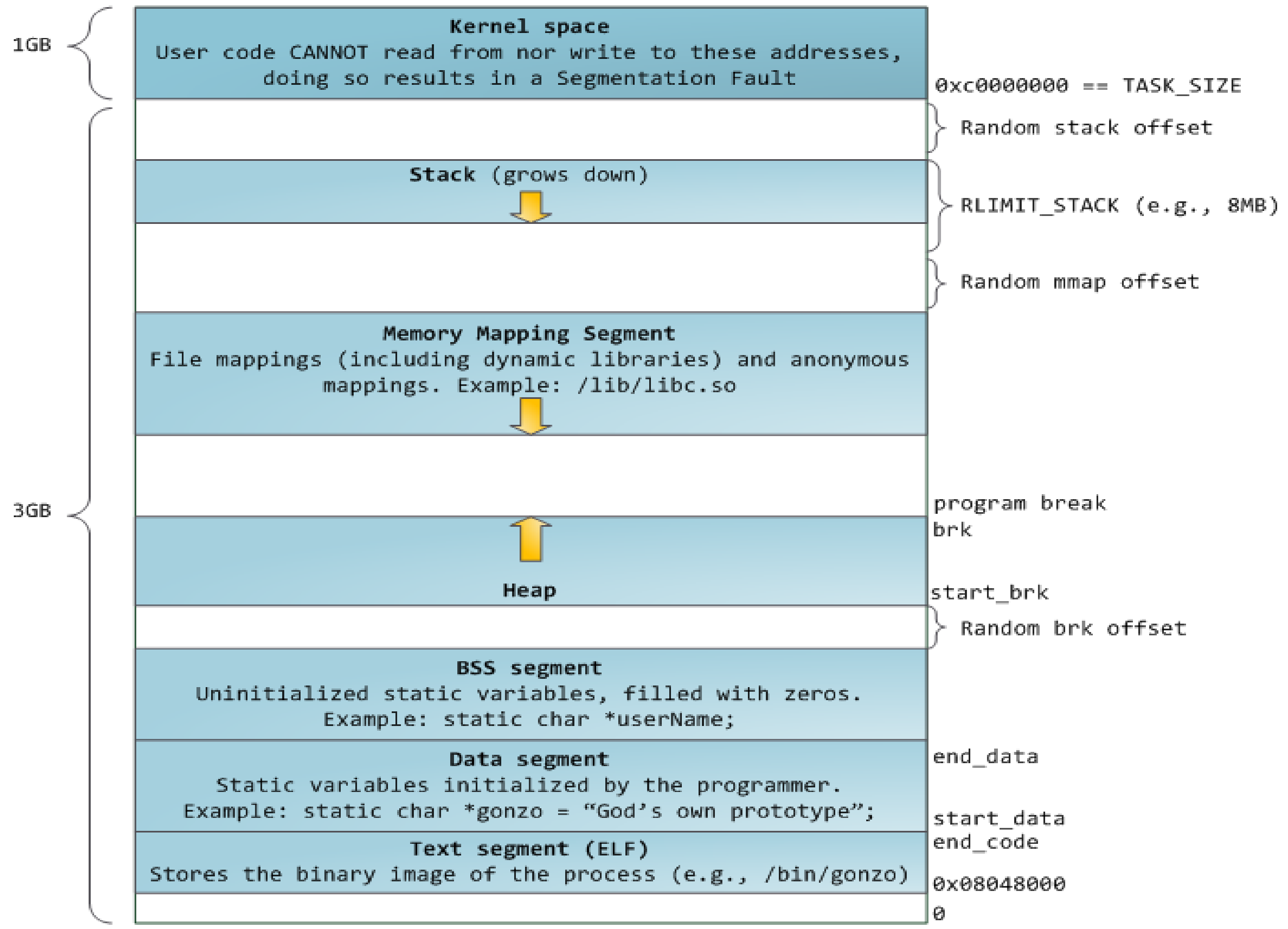
The Linux Kernel Virtual Address Space



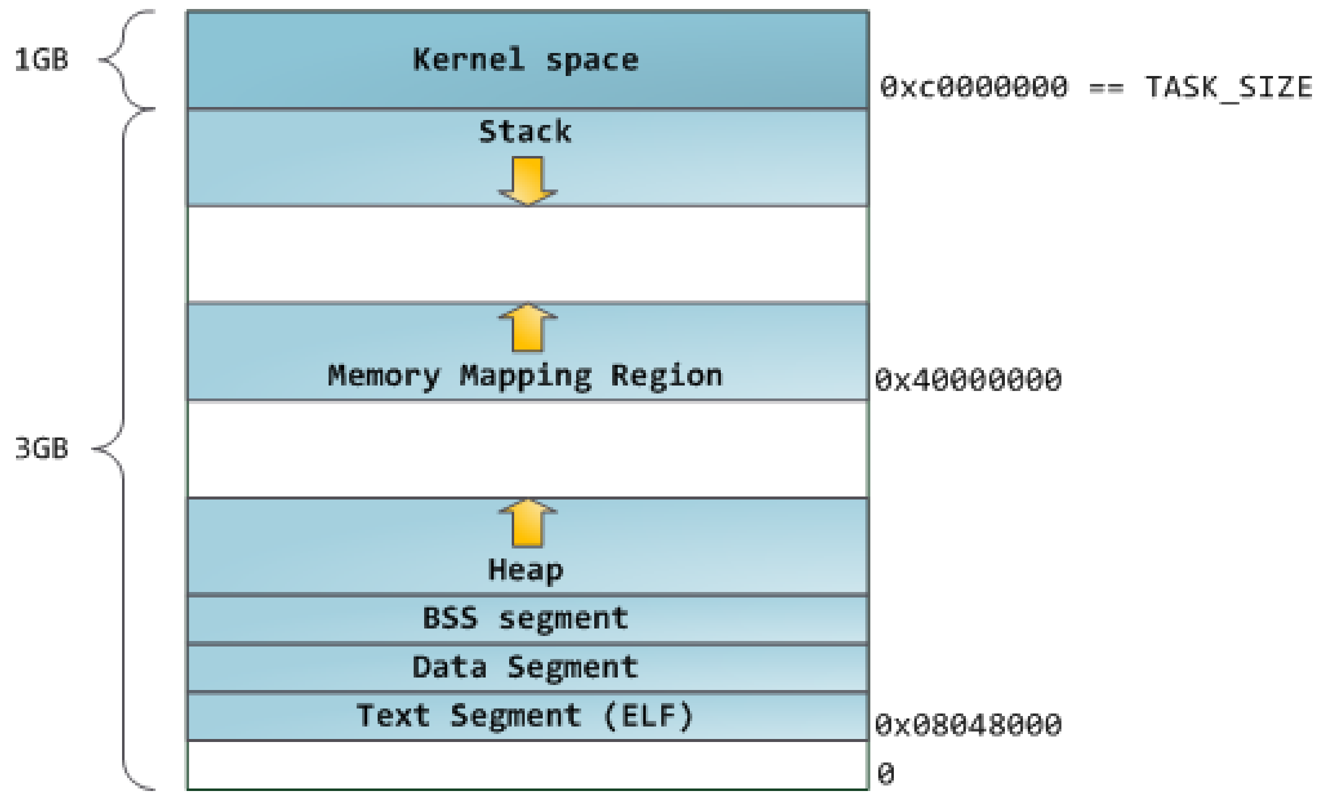
Note 1: The above addresses are configurable – the HIGHMEM start corresponds to a minimum configurable value for high memory and ZONE_NORMAL extending up to 0xF8000000 corresponds to a maximum value.

Note 2: The kernel image can be configured to start elsewhere. Another typical value for instance places it at the start of ZONE_NORMAL, at 0xC1000000

Process Virtual Address Space (flexible layout)


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Process Virtual Address Space (classic layout)



Advantages of the Flexible Layout

Primarily the flexible layout enables:

- Mmap grows upward from 0x40000000 in the classic layout, and down from RLIMIT_STACK + some random offset in the flexible layout
- The new layout is in essence 'self-tuning' the mmap() and malloc() limits: both malloc() and mmap() can grow until all the address space is full. With the old layout, malloc() space was limited to 900MB, mmap() space to ~2GB, as the heap had a hard upper limit of 0x40000000
- Hence, both malloc(), mmap()/shmat() users to utilize the full address space: 3GB on stock x86, 4GB on x86 4:4 or x86-64 running x86 apps.
- The new layout also allows potentially very large continuous mmap()s
- Address space randomization using a random offset is also included in the flexible layout

Sample output from cat /proc/<pid>/maps

```
cat /proc/self/maps
address                   perms  offset       dev    inode     pathname
08048000-08053000 r-xp 00000000 08:07 1048597    /bin/cat
08053000-08054000 r--p 0000a000 08:07 1048597    /bin/cat
08054000-08055000 rw-p 0000b000 08:07 1048597    /bin/cat
08651000-08672000 rw-p 00000000 00:00 0          [heap]
b734b000-b754b000 r--p 00000000 08:07 795175     /usr/lib/locale/locale-archive
b754b000-b754c000 rw-p 00000000 00:00 0
b754c000-b76f0000 r-xp 00000000 08:07 548347     /lib/i386-linux-gnu/libc-2.15.so
b76f0000-b76f2000 r--p 001a4000 08:07 548347     /lib/i386-linux-gnu/libc-2.15.so
b76f2000-b76f3000 rw-p 001a6000 08:07 548347     /lib/i386-linux-gnu/libc-2.15.so
```

Sample output from cat /proc/<pid>/maps

```
cat /proc/self/maps
b76f3000-b76f6000 rw-p 00000000 00:00 0
b770a000-b770b000 r--p 00858000 08:07 795175     /usr/lib/locale/locale-archive
b770b000-b770d000 rw-p 00000000 00:00 0
b770d000-b770e000 r-xp 00000000 00:00 0          [vdso]
b770e000-b7710000 r--p 00000000 00:00 0          [vvar]
b7710000-b7730000 r-xp 00000000 08:07 547269     /lib/i386-linux-gnu/ld-2.15.so
b7730000-b7731000 r--p 0001f000 08:07 547269     /lib/i386-linux-gnu/ld-2.15.so
b7731000-b7732000 rw-p 00020000 08:07 547269     /lib/i386-linux-gnu/ld-2.15.so
bf838000-bf859000 rw-p 00000000 00:00 0          [stack]
```

User-Space Memory Allocation - Malloc

- On demand (page fault first and then actual physical alloc/mapping) – Linux calls this Overcommit
- The flag to change this behaviour:
  - echo 2>/proc/sys/vm/overcommit_memory
- Current malloc version used = ptmalloc2 (lower level routine = _int_malloc)
- What information do we need to have to malloc? Size

User-space memory allocation – Malloc (contd.)

What happens when malloc is called from a process' context the very first time?

- You need to know where free space is within boundaries imposed by system (heap area start, and end of mmap as per flexible layout)
  - ✓ For this you need to first look at memory hierarchy: Page-level and block level
  - ✓ What is the minimum block size? – 16 bytes (8 bytes + header-footer overhead) (implementation dependent)
  - ✓ Malloc (4), malloc (8) both return 16 bytes, malloc (12), malloc (16) both return 24 bytes – multiples of 8

User-space memory allocation – Malloc (contd.)

- Does malloc need to return a virtually contiguous space? Too much space overhead otherwise (next pointer etc.) and temporal locality might be lost as well, but most importantly, the compiler relies on the fact that the virtual space is contiguous.

- The physical non-contiguousness is the basis of paging virtual memory, and results in zero external fragmentation at the page level. Plus, it facilitates swap which the kernel space doesn’t need to.

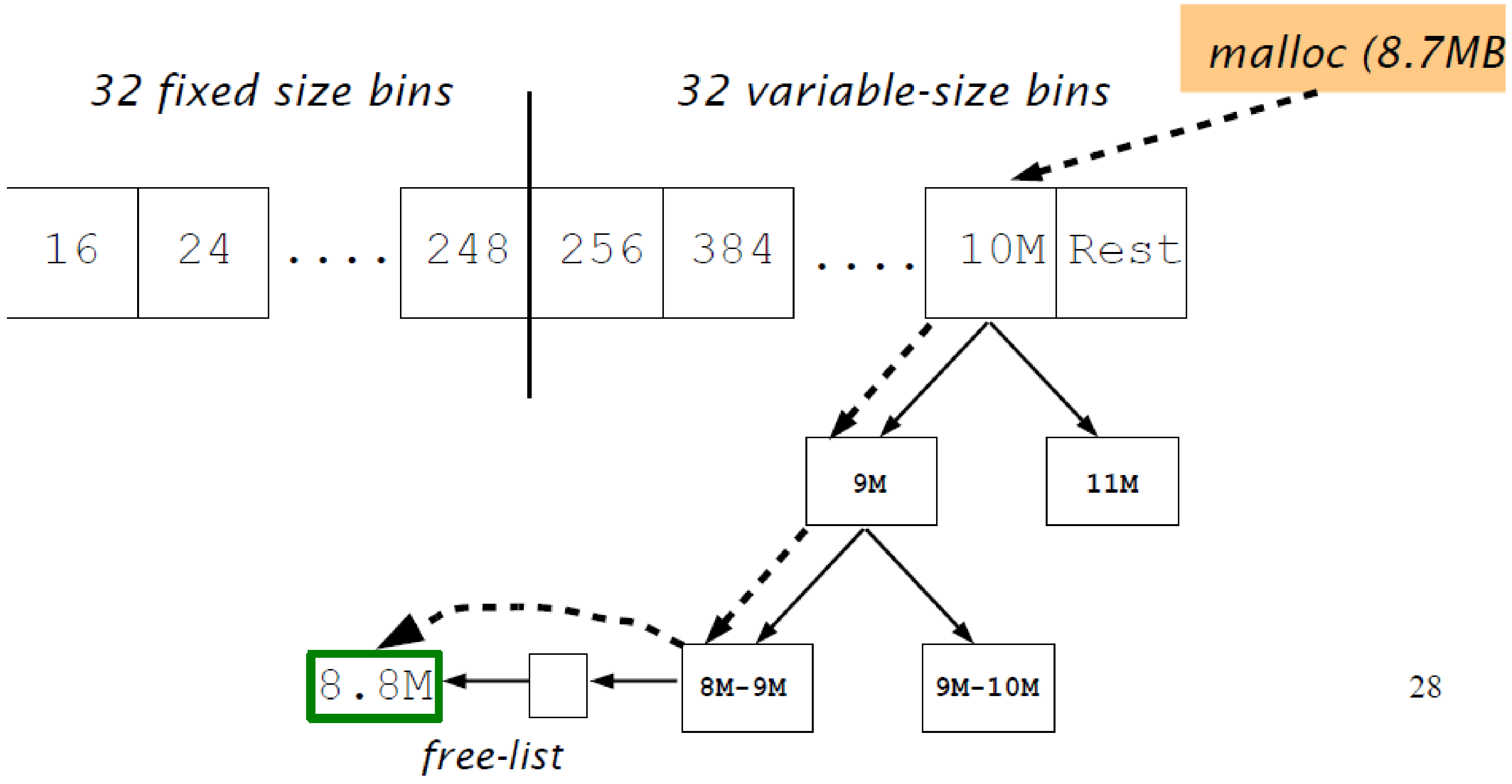
Malloc (contd.)

- Best-fit, first-fit: Best-fit is time-consuming, but results in lesser external fragmentation for bigger-size blocks
- Sbrk/mmap to increase heap size if available virtual memory is not sufficient on the heap
- Fast bin size (very small) (< 64 bytes) – bins/free lists of different but fixed sizes, faster than best fit from a heterogeneous mix. Minimum search. When do blocks/chunks become part of bins? On every free. No coalescing in interests of speed.
- Unsorted bin ??

Malloc (contd.)

- Small size (>= 64, < 512 bytes) – best-fit with coalescing.
- Large size (>= 512, < mmap threshold (128 KB) - variable size bins with trees /optimal-for-search data structures whose nodes are sorted by size – bigger range, lesser number of bins, more searching.
- Very large (> mmap threshold 128 KB) – relies on mmap mapping facilities and free now actually returns memory to the operating system by calling munmap. The tradeoff is that more minor faults occur on subsequent big malloc's as pages are freed.
- For cache line alignment, cache-line aligned alloc routines, posix_memalign.
- Can malloc-ed memory sit astride a page boundary? No, because when you run out of user space memory, you mmap/sbrk and go into the next page completely, no piece-by-piece thing.

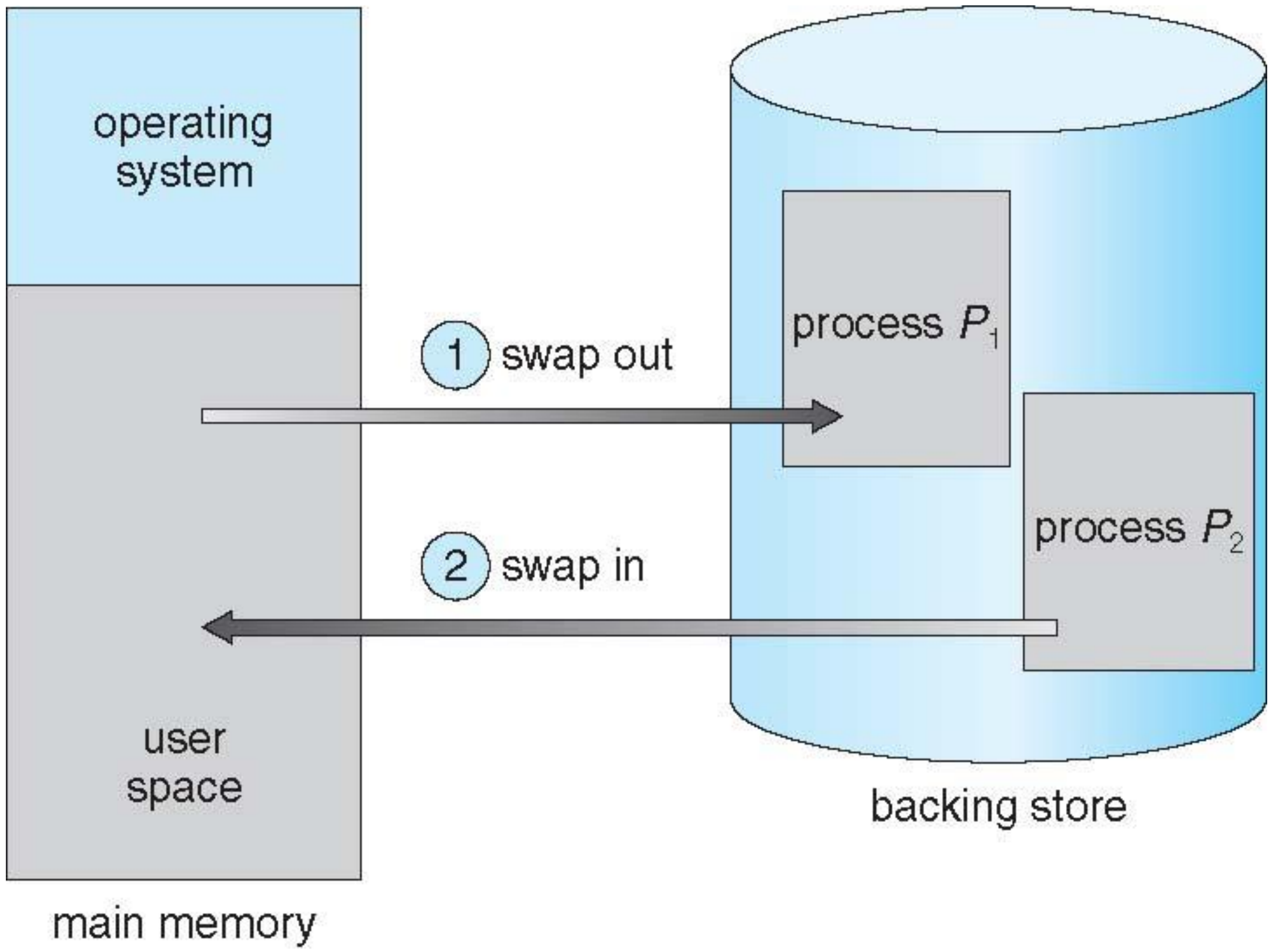
Malloc – Binning



Malloc (contd.) – A reading list

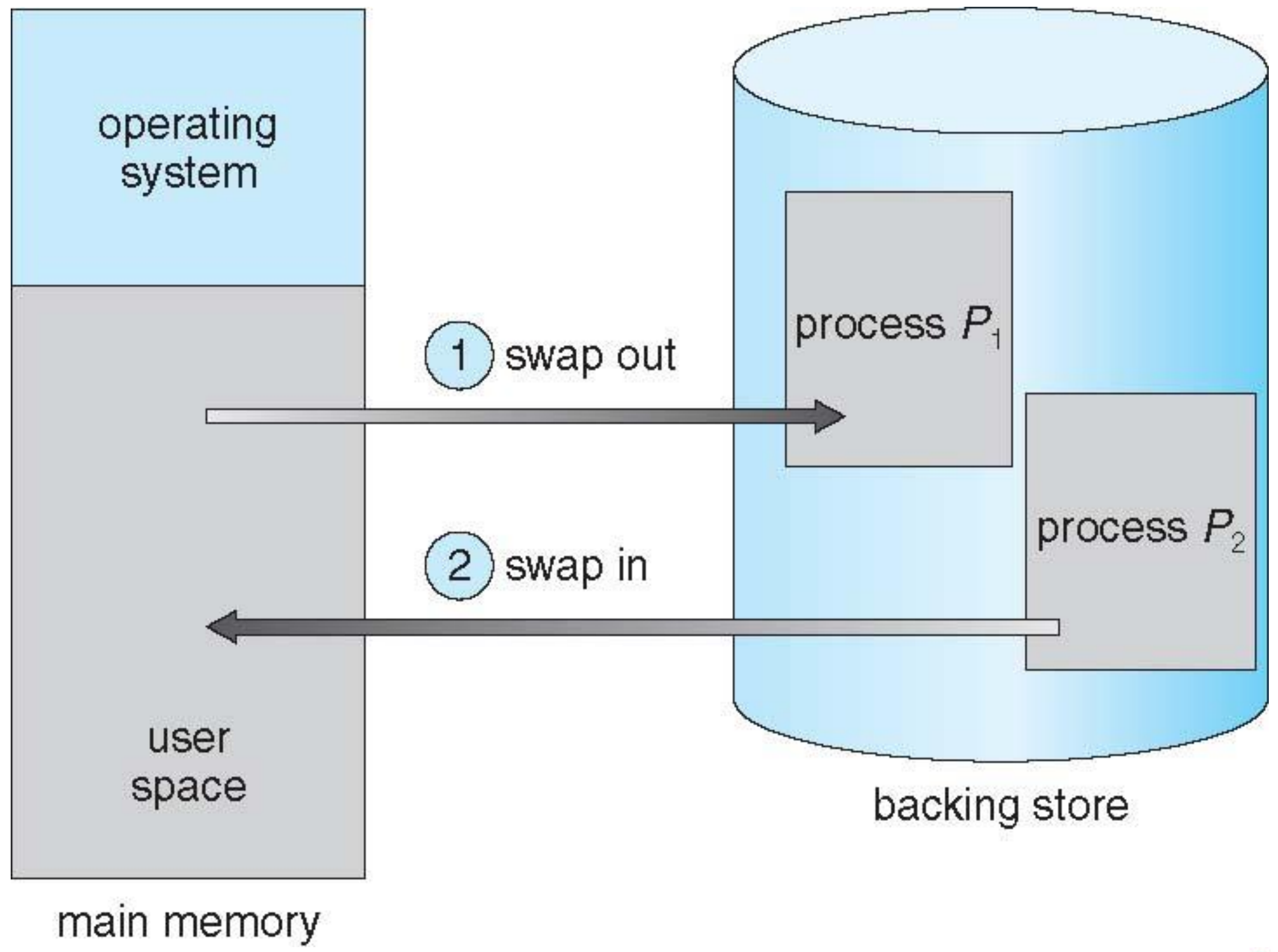
- The additional complexity of process address spaces
  - ✓ Mapping and creation of physical spaces/pages only on demand, i.e. on page faults
- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
- http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf
- http://files.cnblogs.com/files/hseagle/demo.pdf
- http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_3.html
- http://web.eecs.utk.du/~plank/plank/classes/cs360/360/notes/Malloc1/lecture.html
- http://www.eecs.harvard.edu/~mdw/course/cs61/mediawiki/images/5/51/Malloc3.pdf (Binning diagram courtesy this link)
- ftp://g.oswego.edu/pub/misc/malloc.c
- http://en.wikibooks.org/wiki/C_Programming/C_Reference/stdlib.h/malloc

Swapping


operating system
1 swap out
2 swap in
process P_1
process P_2
user space
main memory
backing store

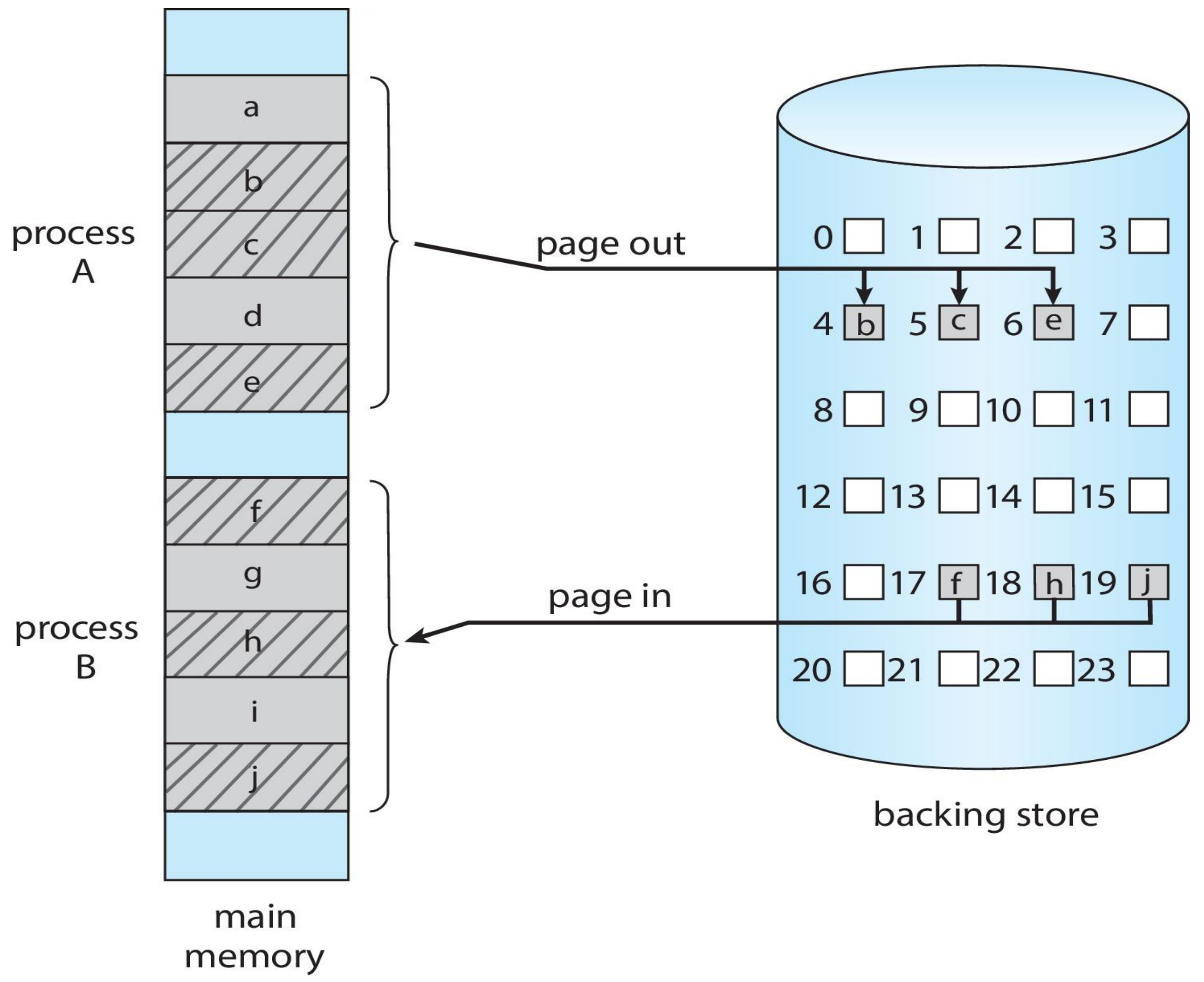
Schematic View of Swapping

What swapping used to mean originally

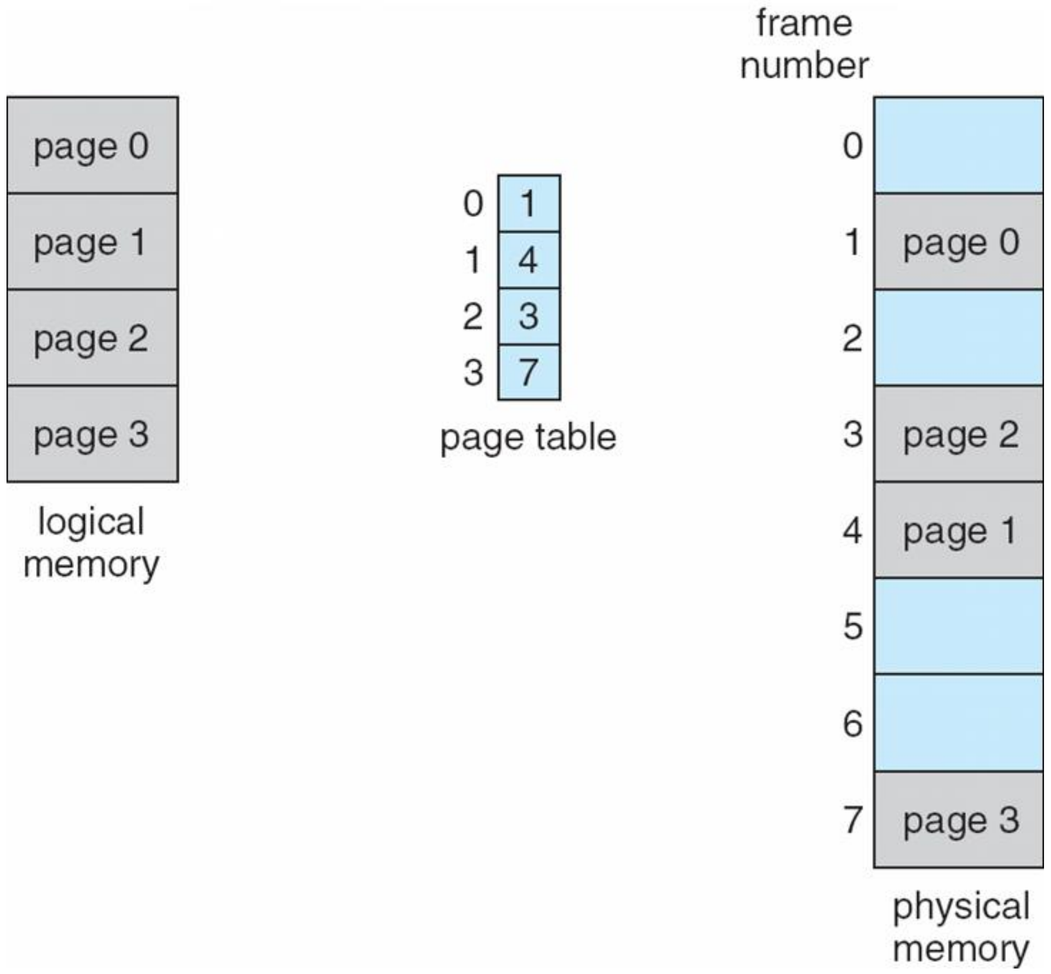


Swapping with Paging

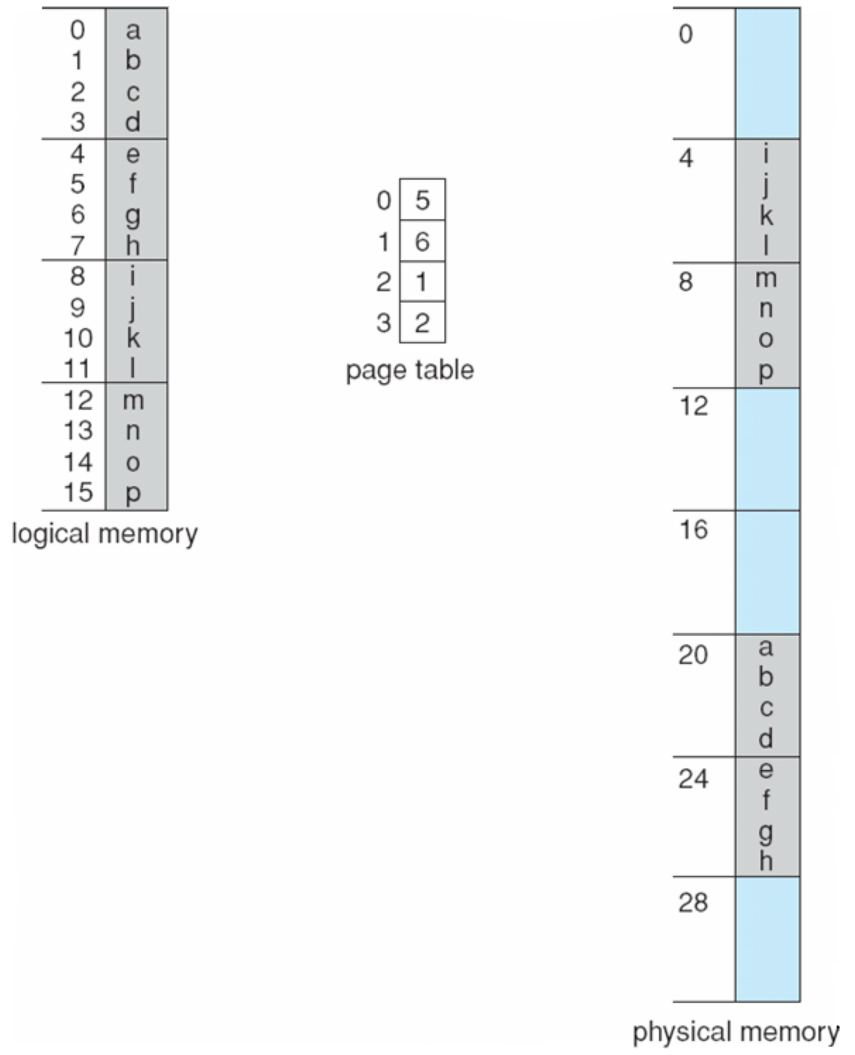
What swapping means in modern operating systems



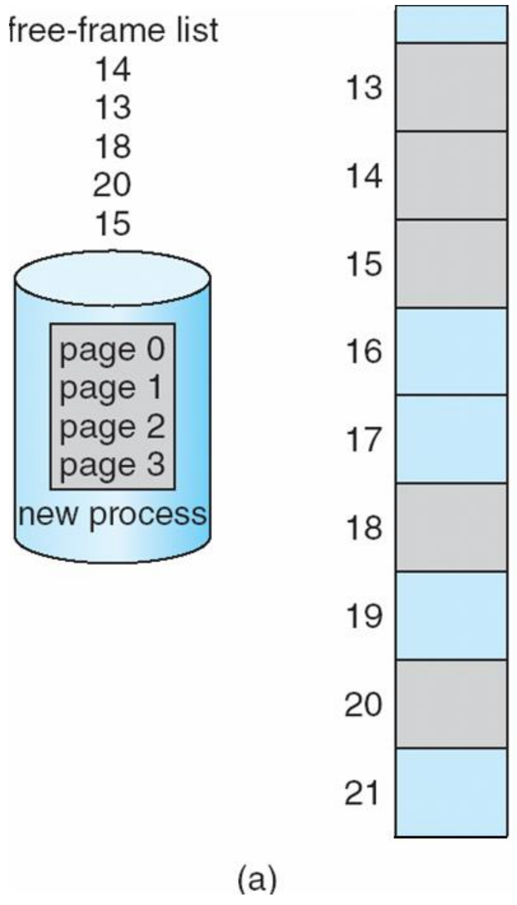
Paging model of logical and physical memory



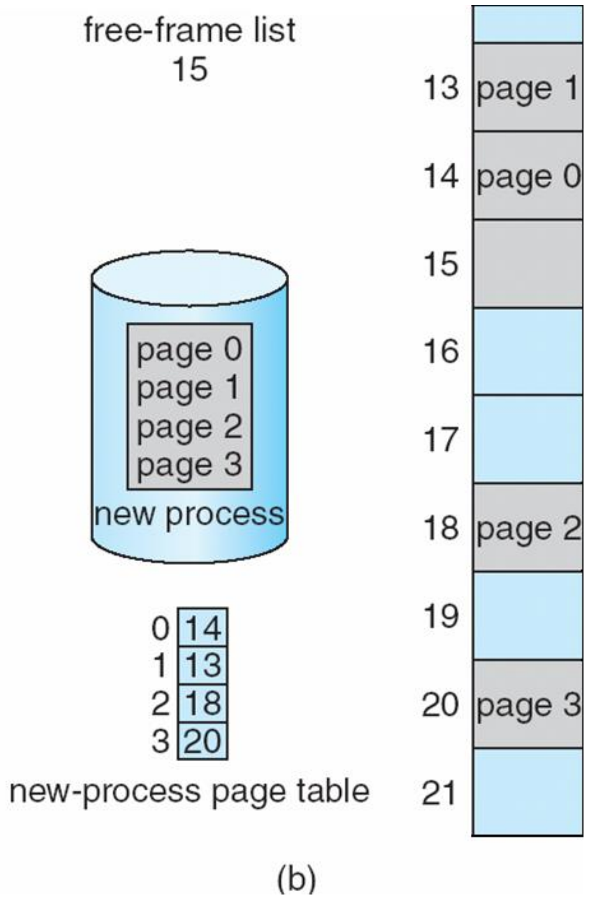
Paging example (with values loaded in memory locations)



New process physical page allocation when there are free pages



Before allocation



After allocation

Page faults – Basic Premise

- The previous slides serve as a pictorial view of a simple swapping scheme where free frames are marked and can be used as a target a new process physical page allocation
- That’s simple enough
- Now, let’s think of a scenario, where there are not enough free pages for use by a new process

Fault handling/demand paging of user-space pages

Minor fault:

- So-far unmapped page, PTE all zeroes, find virtual address in memory management kernel data struct
  - ✓ If found, then allocate space for PTE, get physical page from free list, set PTE to point the relevant virtual address to the physical page.
  - ✓ If not found, cause segmentation fault/trap and cause the process to exit

Fault handling/demand paging of user-space pages

Major fault:

- PTE found, non-zero entry, but present bit reset. Find swap index from PTE, swap page in after finding target physical slot, and swap out of target
- Swap-out involves cycling through reverse mapping to figure out set of logical pages mapped to the target
- Swap-in in involves interacting with the bio subsystem and device driver to get the page contents
- Target physical slot – LRU, accessed bit in PTE, periodically reset by software, set on access by hardware to implement LRU
- Note: Only anonymous pages get swapped out – the file page cache is flushed and shrunk, so will have zeroes in its PTE

Thank You!