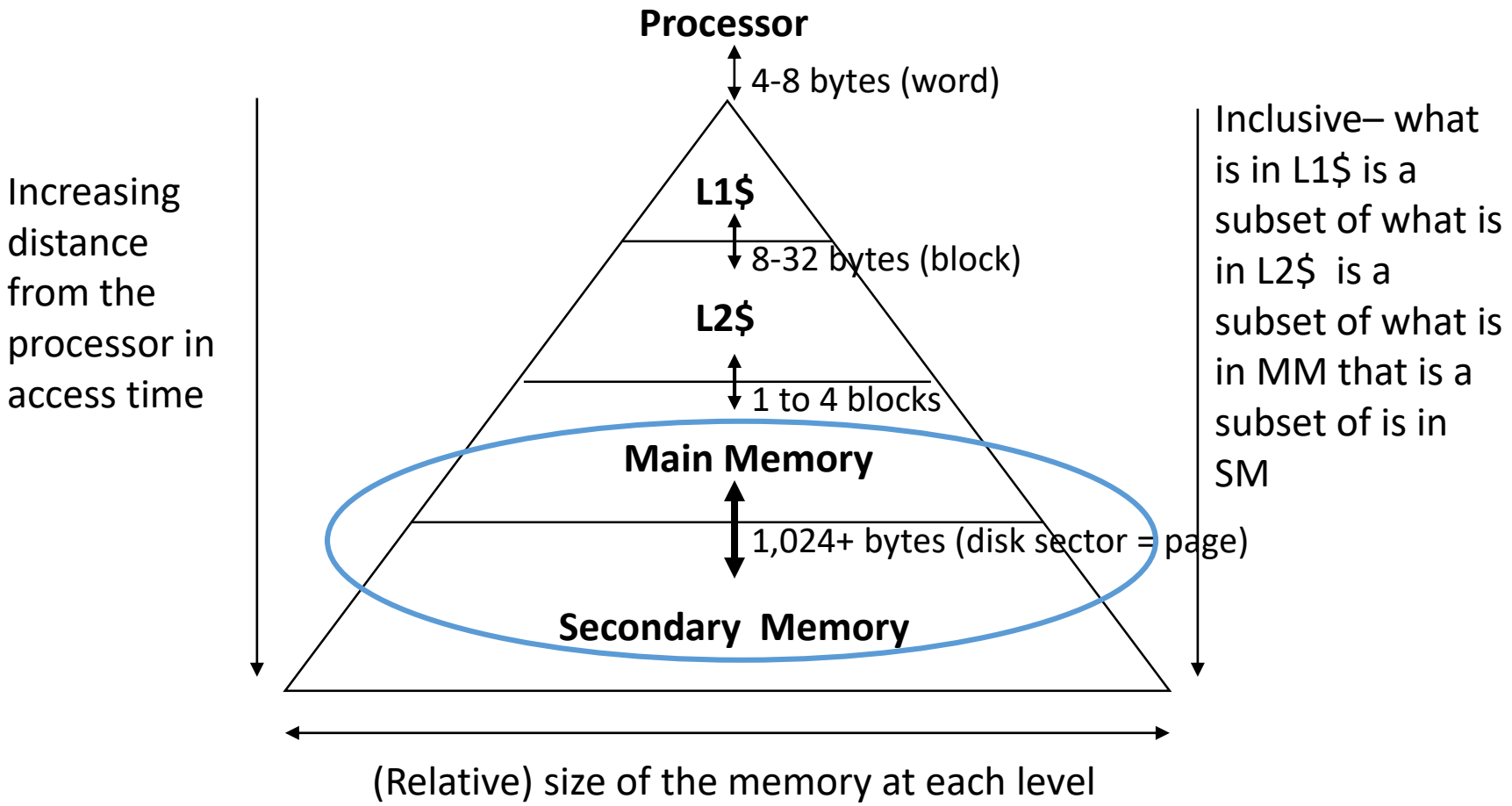
Lecture 8

Memory Management

The Memory Hierarchy



Virtual Memory

- Use main memory as a “cache” for secondary memory
  - Allows efficient and safe sharing of memory among multiple programs
  - Provides the ability to easily run programs larger than the size of physical memory
  - Simplifies loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)
- What makes it work? – the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a “virtual” address space
  - During run-time each virtual address must be translated to a physical address (an address in main memory)

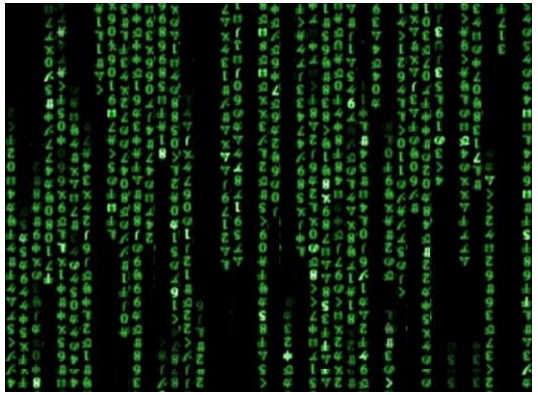
A physically addressed machine

Apple][(1977)

RAM Complement	Apple II System
4K	$ 1,298.00
48K	2,638.00

RAM Organization and Usage

Page # Dec	Page # Hex	Used for:	
0	$00	System Programs	
1	$01	System Stack	
2	$02	Input Buffer	
3	$03	Monitor Vector Locations	
4	$04		
5	$05	Tex/Lo-Res Graphics	
6	$06	Primary Page Storage	
7	$07		
8	$08		
9	$09	Text/Lo-Res Graphics	
10	$0A	Secondary Page Storage	FREE RAM
11	$0B		
12	$0C		
through		FREE RAM	
31	$1F		
32	$20	Hi-Res Graphics	
through		Primary Page	FREE RAM
63	$3F	Storage	
64	$40	Hi-Res Graphics	
through		Seconary Page	FREE RAM
95	$5F	Storage	
96	$60		
through		FREE RAM	
191	$BF		
192	$C0	I/O and softswitches	
193	$C1		
through		I/O shared ROM space	
199	$C7		

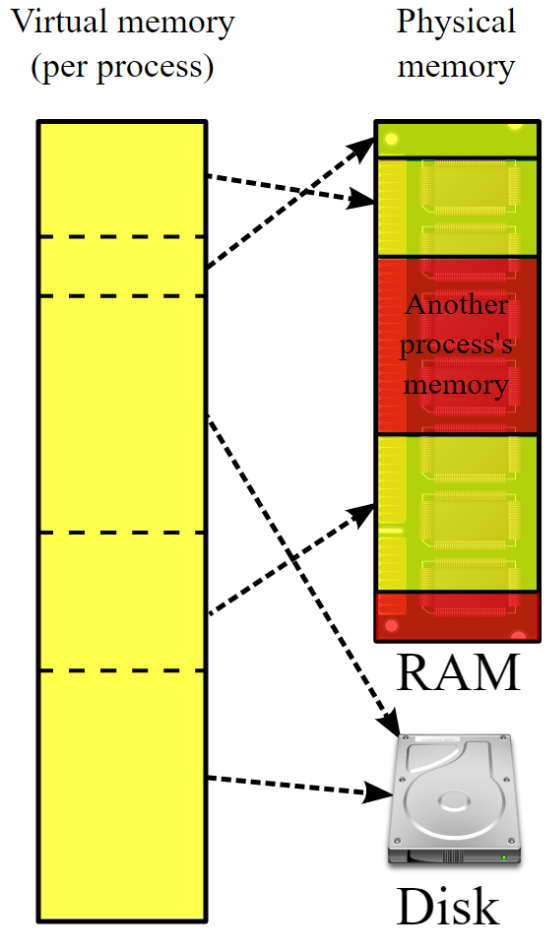


Using physical addressing

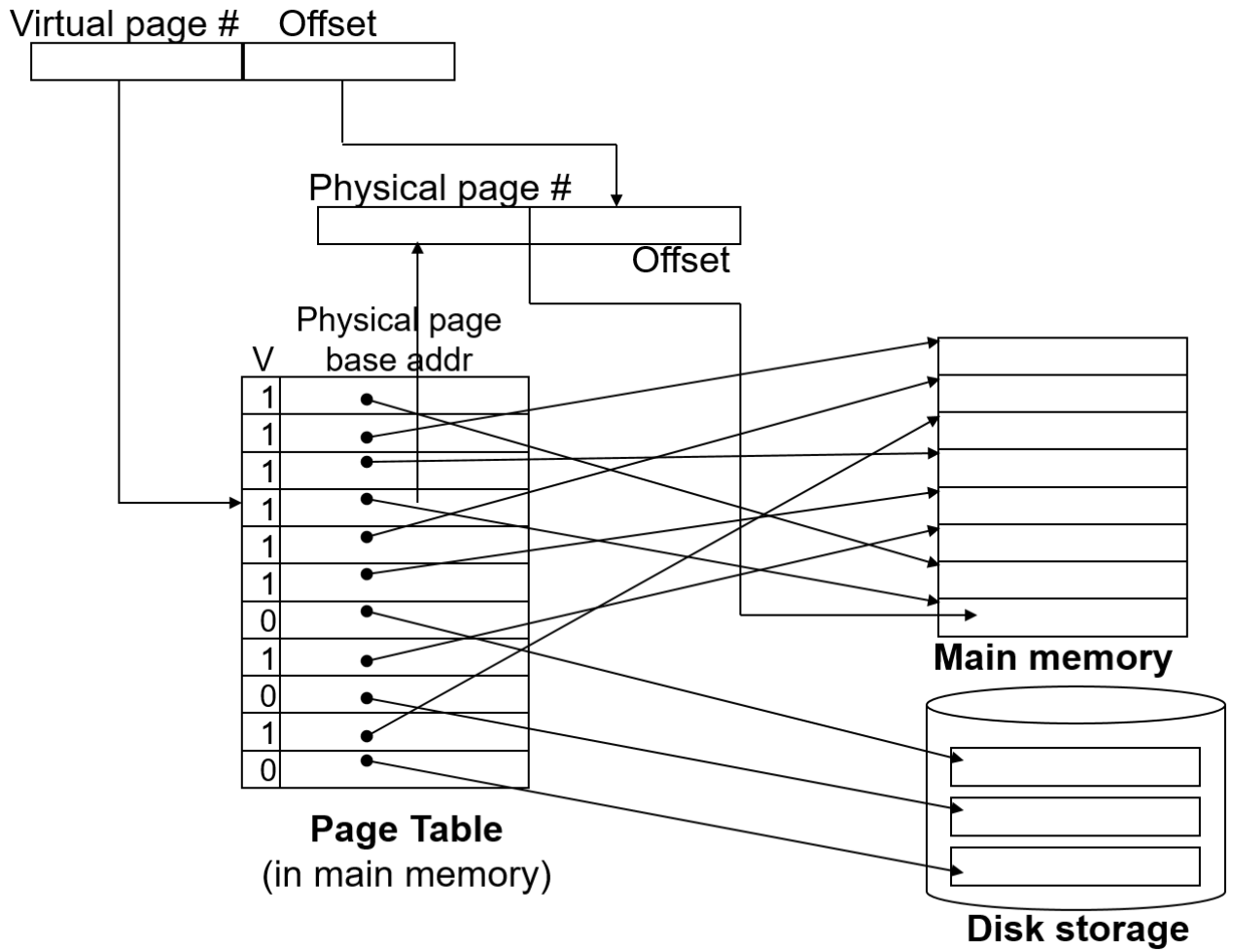
- All programs share one address space: the physical address space
- Machine language programs must be aware of the machine organization
- No way to prevent a program from accessing any machine resource

The solution: virtual addressing

- User programs run in an standardized virtual address space
- Address Translation hardware managed by the operating system (OS) maps virtual address to physical memory
- Hardware supports “modern” OS features:
  - Protection,
  - Translation,
  - Sharing



Address Translation Mechanism



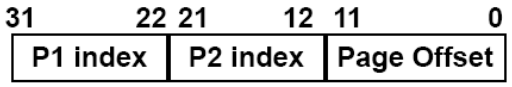
Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

Each process needs its own address space!

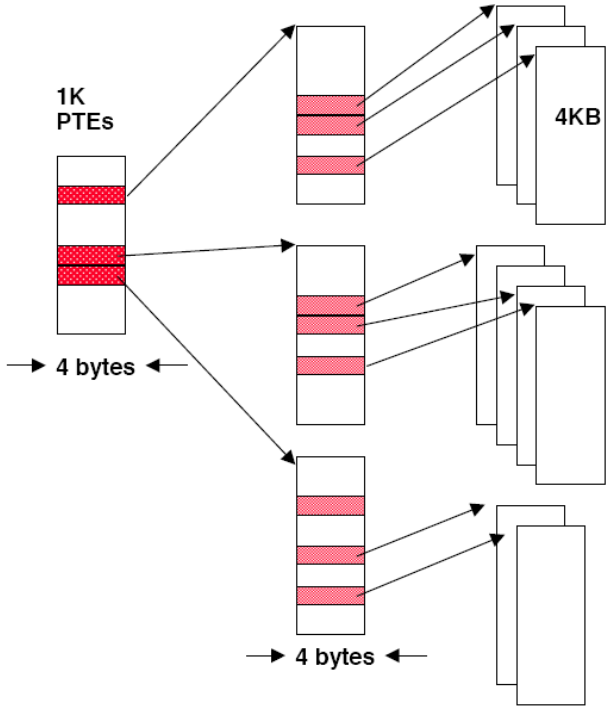
Two-level Page Tables

32 bit virtual address



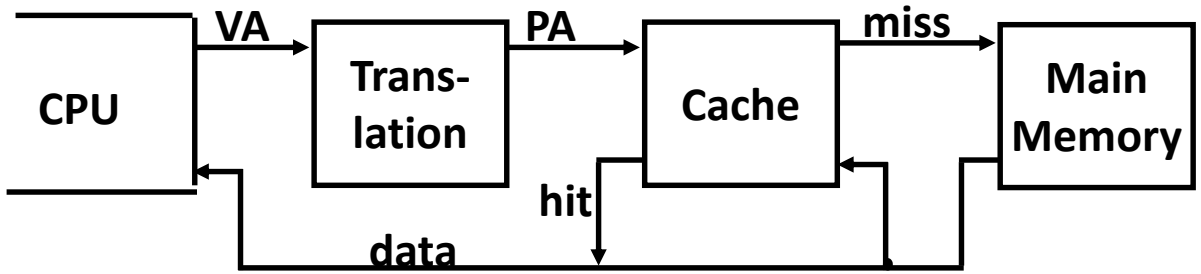
Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated



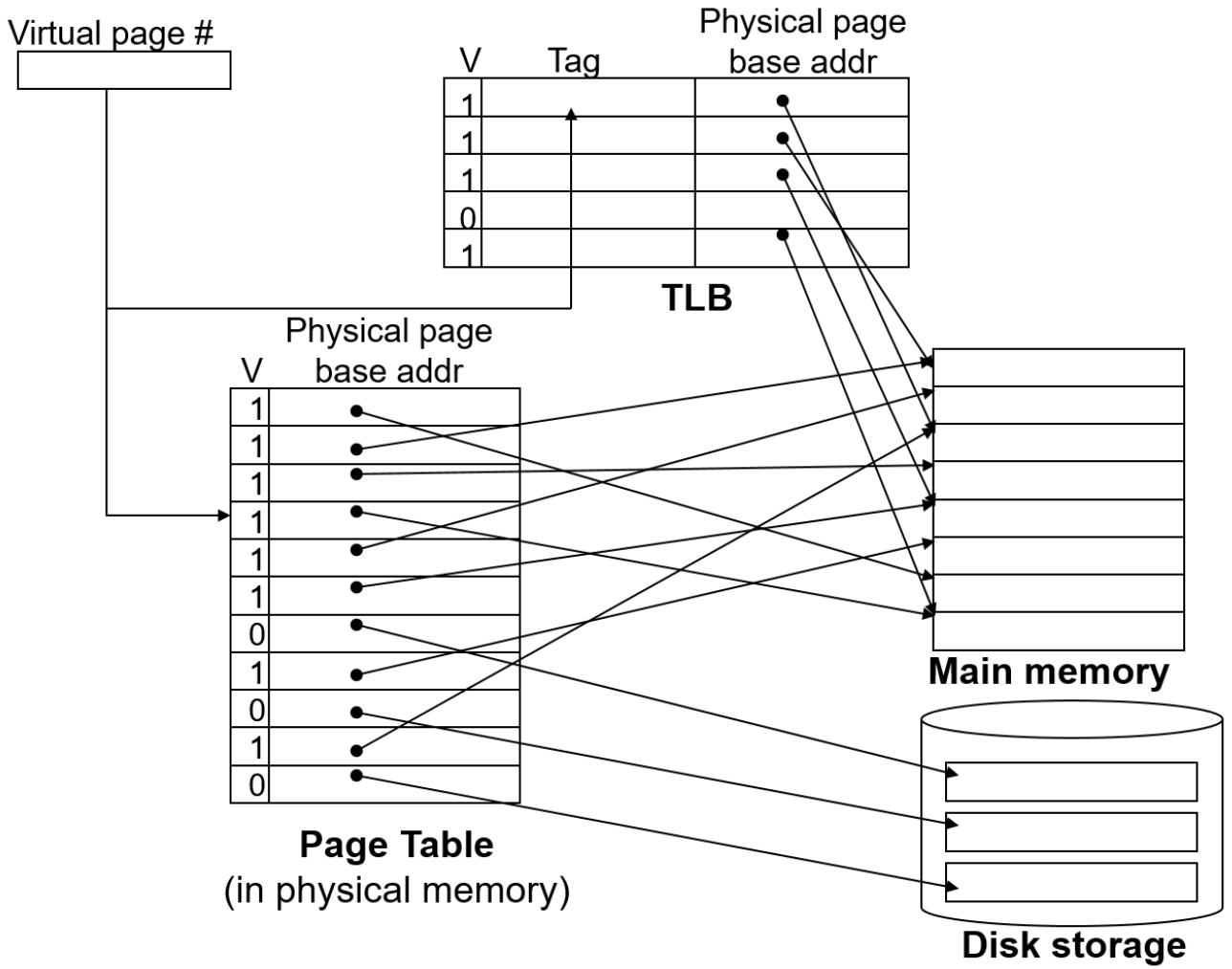
Virtual Addressing with a Cache

- Thus it takes an *extra* memory access to translate a VA to a PA



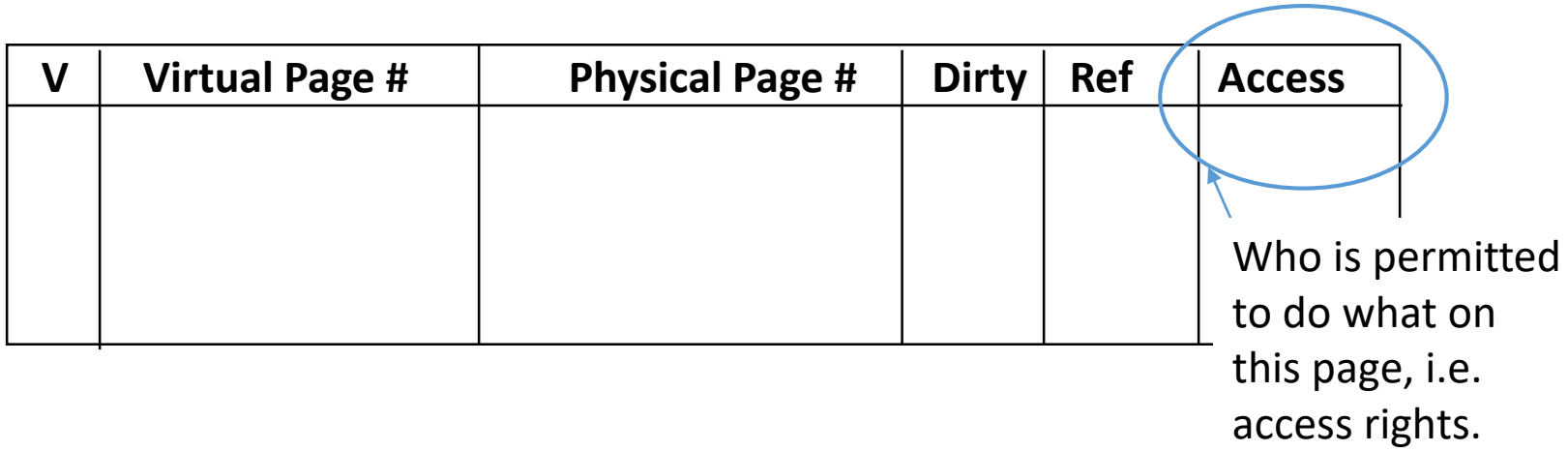
- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)
- The hardware fix is to use a Translation Lookaside Buffer (TLB) – a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup

Making Address Translation Fast



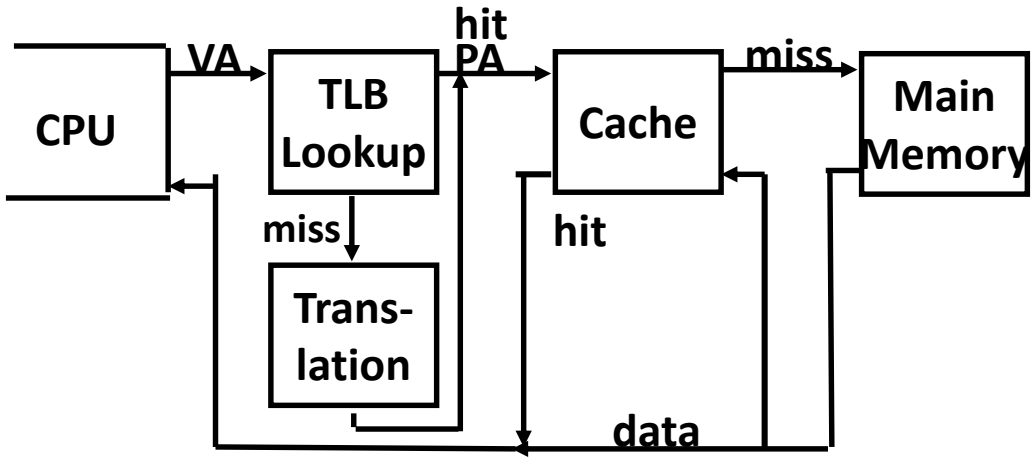
Translation Lookaside Buffers (TLBs)

- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped



- TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)
  - TLBs are typically not more than 128 to 256 entries even on high end machines

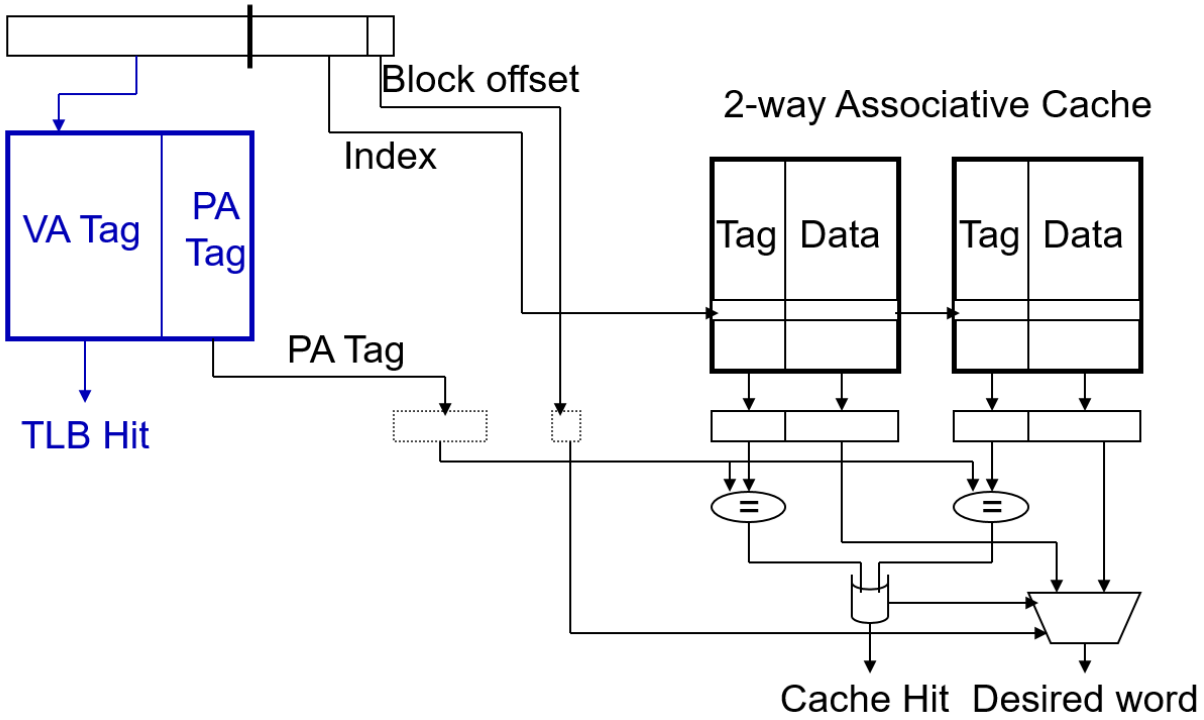
A TLB in the Memory Hierarchy



- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
    - Takes 10's of cycles to find and load the translation info into the TLB
  - If the page is not in main memory, then it's a true page fault
    - Takes 1,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults

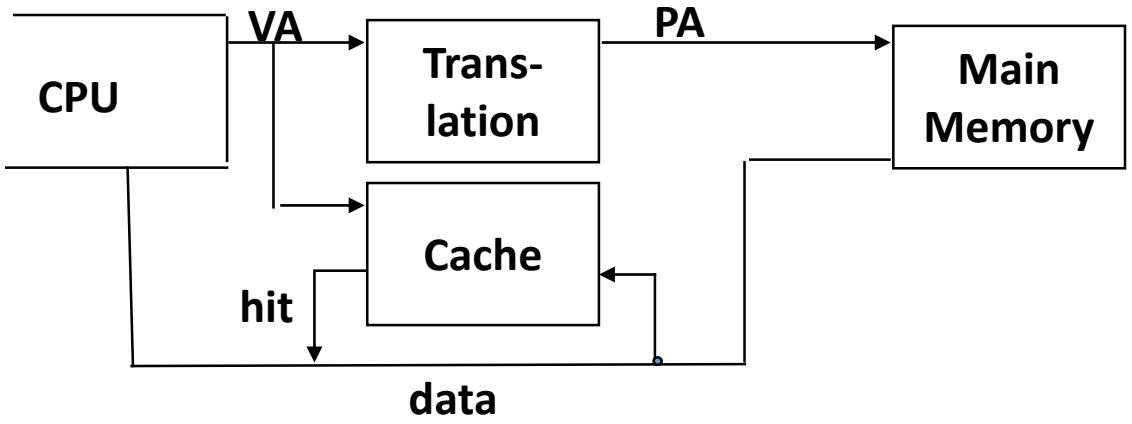
Reducing Translation Time

- Can overlap the cache access with the TLB access
  - Works when the high order bits of the VA are used to access the TLB while the low order bits are used as index into cache



Why Not a Virtually Addressed Cache?

- A virtually addressed cache would only require address translation on cache misses



but

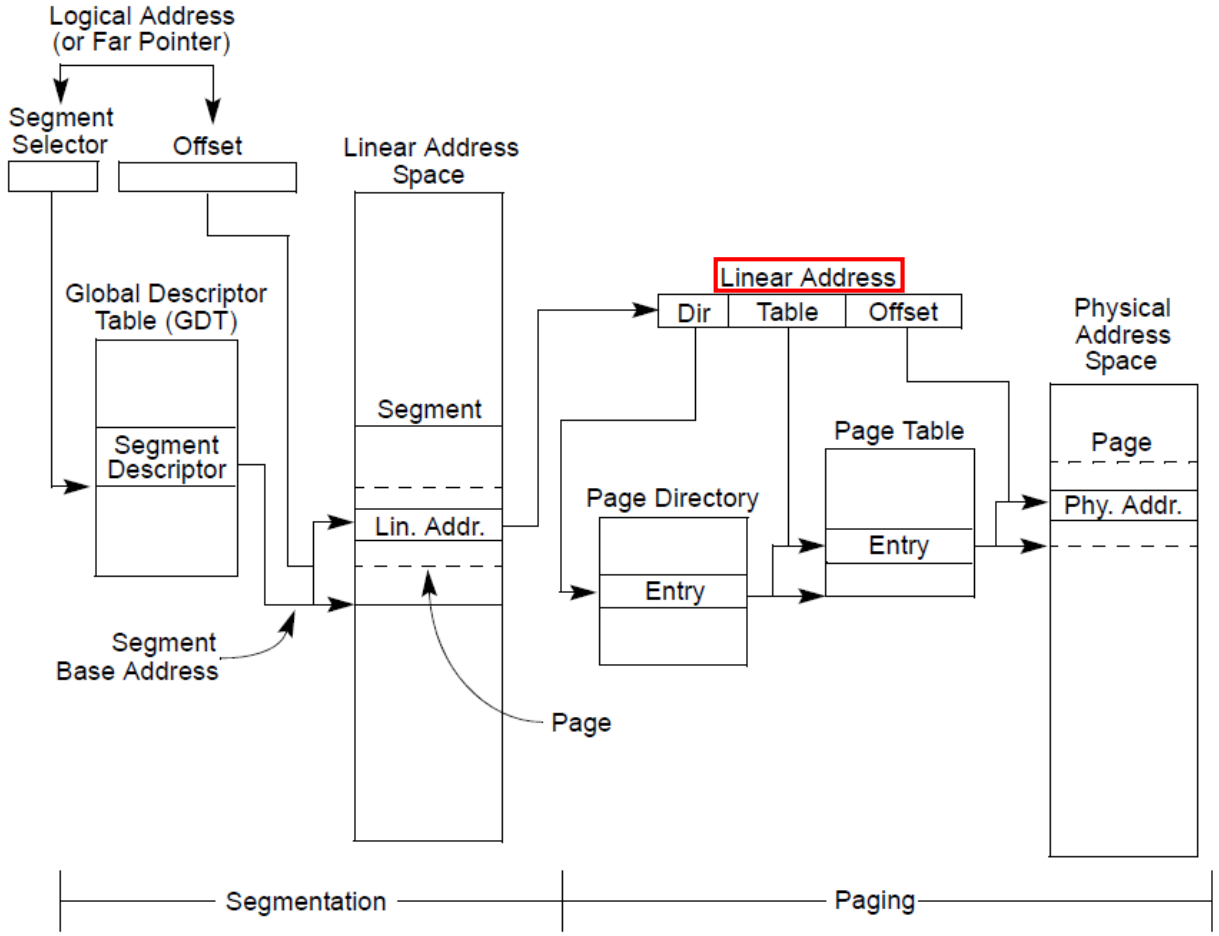
- Two different virtual addresses can map to the same physical address (when processes are sharing data), i.e., two different cache entries hold data for the same physical address – synonyms
  - Must update all cache entries with the same physical address or the memory becomes inconsistent

The Hardware/Software Boundary

- What parts of the virtual to physical address translation is done by or assisted by the hardware?
  - Translation Lookaside Buffer (TLB) that caches the recent translations
    - TLB access time is part of the cache hit time
    - May allot an extra stage in the pipeline for TLB access
  - Page table storage, fault detection and updating
    - Page faults result in interrupts (precise) that are then handled by the OS
    - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the Page Tables
  - Disk placement

Memory in the Linux Kernel

Memory addressing – Segmentation and Paging



Getting segmentation out of the way

- Only use segmentation when it has to
- Addresses are linear
- Make 0x00000000 the base of all “segments”
- “True” address is in the offset

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

Paging in Linux-x86

- (Contiguous) linear addresses are grouped in fixed length intervals call pages
  - On x86, a page is 4KB (with later extension to support larger pages)
- Physical memory is partitioned into fixed length page frames (or simply called *frames*)
  - One page frame contains a single page
- Linear addresses are translated to physical addresses using page tables
- Page is turned on in x86 by setting the **`PG`** bit in **`CR0`**
- The root page table is pointed to by the page directory base register field in **`CR3`**

Paging in 64 bits

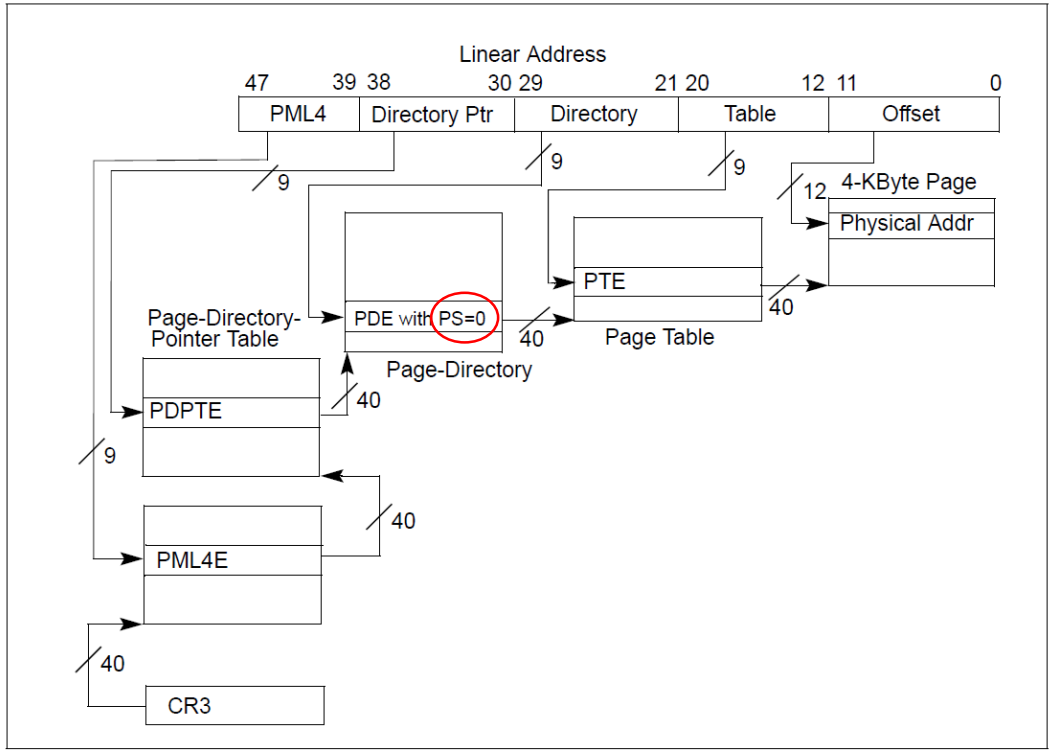


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

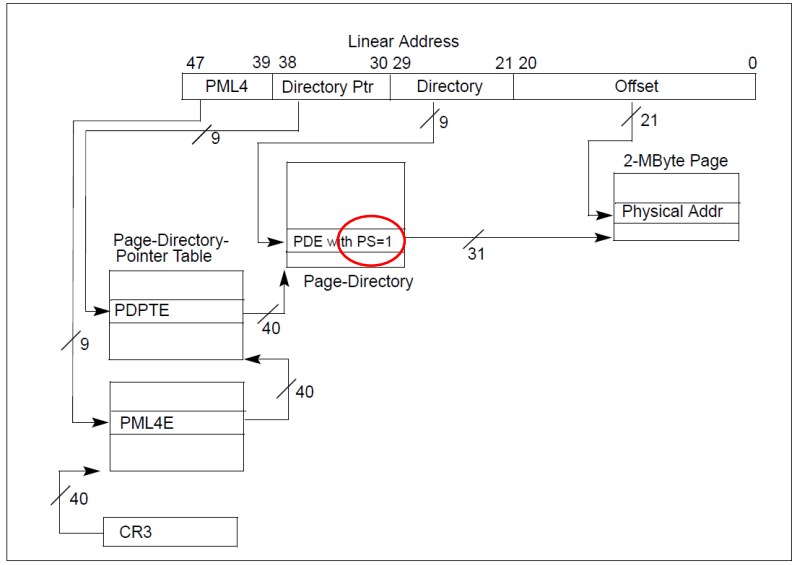


Figure 4-9. Linear-Address Translation to a 2-MByte Page using IA-32e Paging

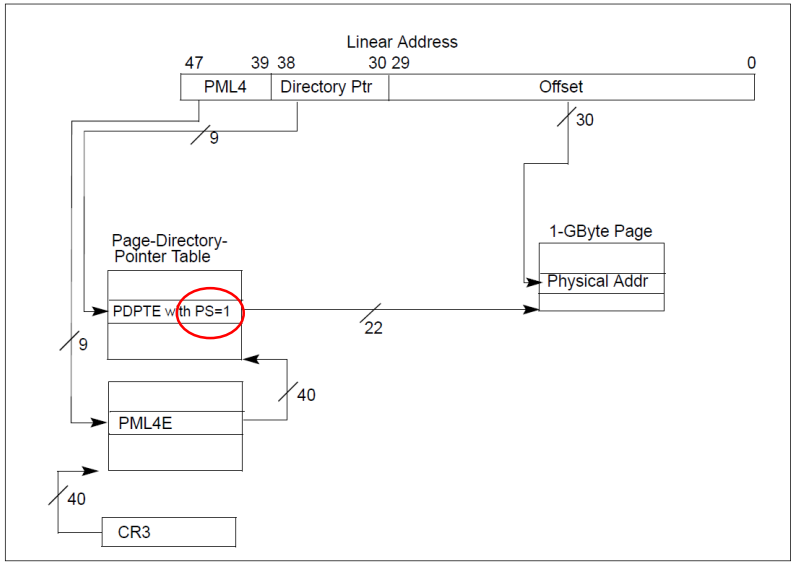


Figure 4-10. Linear-Address Translation to a 1-GByte Page using IA-32e Paging

Four level page tables

- **The page map level 4 (PML4)** — An entry in a PML4 table contains the physical address of the base of a page directory pointer table, access rights, and memory management information. The base physical address of the PML4 is stored in CR3.
- **A set of page directory pointer tables** — An entry in a page directory pointer table contains the physical address of the base of a page directory table, access rights, and memory management information.
- **Sets of page directories** — An entry in a page directory table contains the physical address of the base of a page table, access rights, and memory management information.
- **Sets of page tables** — An entry in a page table contains the physical address of a page frame, access rights, and memory management information.

CR3

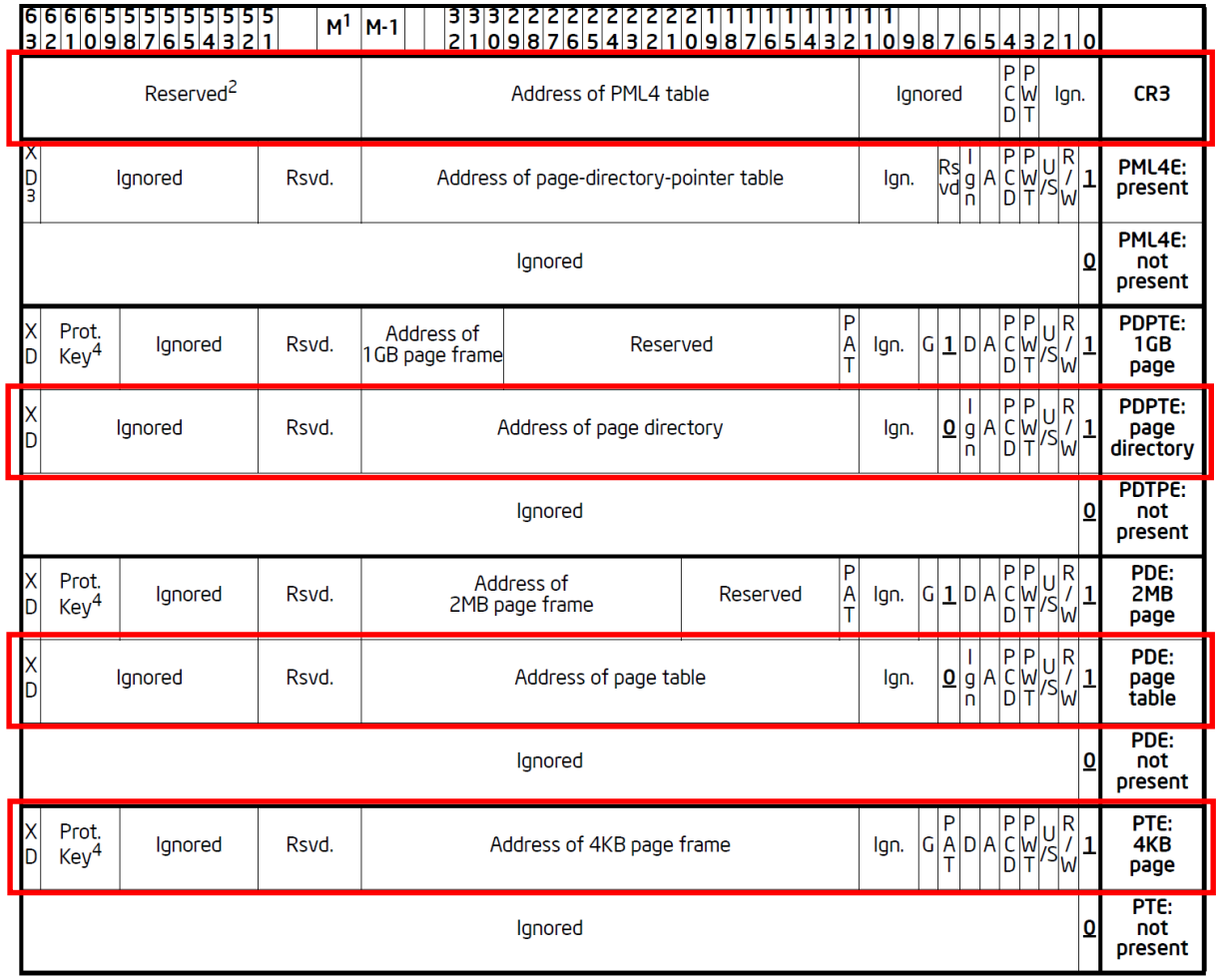
Table 4-13. Use of CR3 with IA-32e Paging and CR4.PCIDE = 1

Bit Position(s)	Contents
11:0	PCID (see Section 4.10.1)[1]
M–1:12	Physical address of the 4-KByte aligned PML4 table used for linear-address translation[2]
63:M	Reserved (must be 0)[3]

Process Context ID (PCID)

- In other processors also known as the address space ID (ASID)
- Enable faster context switching
- Used to distinguish one process' address space from another
  - 12 bits in length
  - Affects how TLB works
  - Value of 0 reserved for various condition checking
- Linux use it on a per-CPU basis instead of a global ASID

The Table Entries



63	62–M	M-1	32–12	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	Prot. Key[4]	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	Prot. Key[4]	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	Prot. Key[4]	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

PML4's 64 bit entry

Table 4-14. Format of an IA-32e PML4 Entry (PML4E) that References a Page-Directory-Pointer Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page-directory-pointer table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 512-GByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 512-GByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Reserved (must be 0)
11:8	Ignored
M–1:12	Physical address of 4-KByte aligned page-directory-pointer table 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 512-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

M = Max physical address length supported by this processor

Page Directory Pointer Table Entry

Table 4-16. Format of an IA-32e Page-Directory-Pointer-Table Entry (PDPTE) that References a Page Directory

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
1 (R/W)	Read/write; if 0, writes may not be allowed to the 1-GByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 1-GByte page; see Table 4-15)
11:8	Ignored
(M–1):12	Physical address of 4-KByte aligned page directory 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 1-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Page Directory Entry

Table 4-18. Format of an IA-32e Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 4-17)
11:8	Ignored
(M–1):12	Physical address of 4-KByte aligned page table 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 2-MByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Page Table Entry for 4K Pages

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

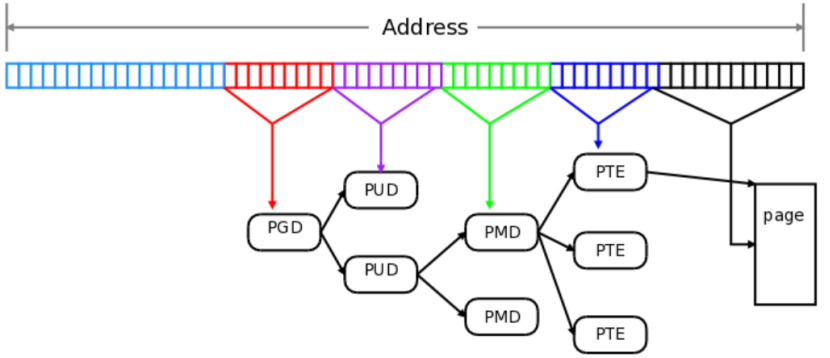
Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see 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)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1, determines the protection key of the page (see Section 4.6.2); ignored otherwise
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)

PTE protection and status bits

Bit	Function
_PAGE_PRESENT	Page is resident in memory and not swapped out
_PAGE_PROTNONE	Page is resident but not accessable
_PAGE_RW	Set if the page may be written to
_PAGE_USER	Set if the page is accessible from user space
_PAGE_DIRTY	Set if the page is written to
_PAGE_ACCESSED	Set if the page is accessed

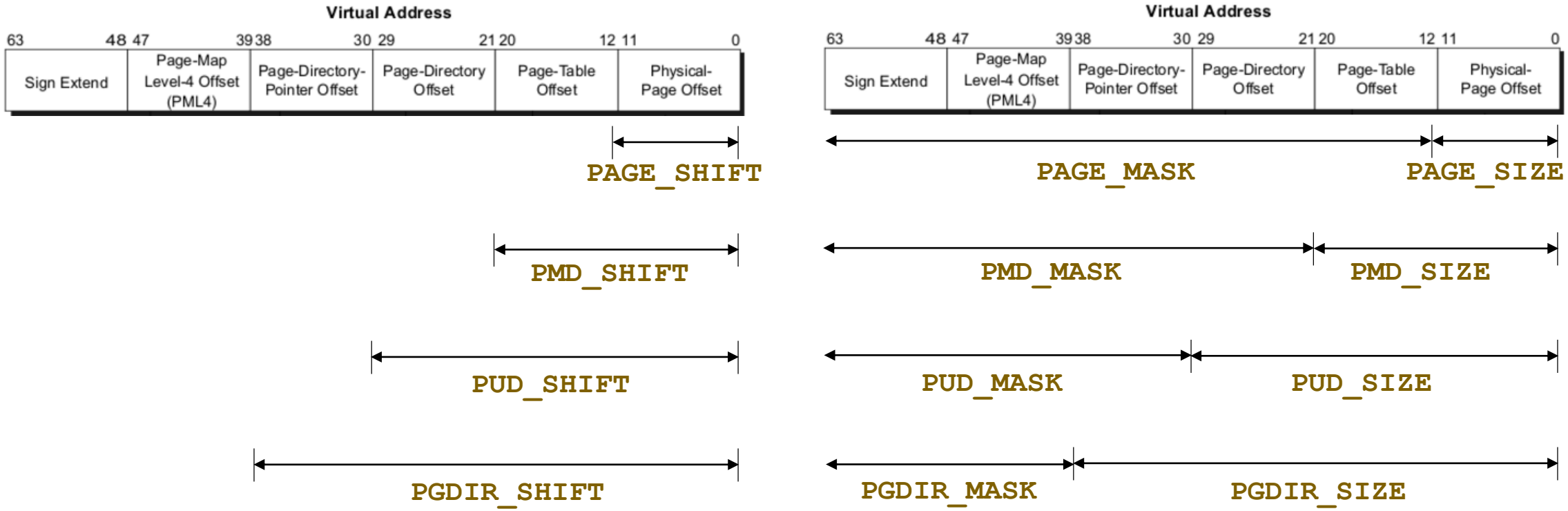
Linux and x86-64 paging

- Linux supports 4 level paging:
  - Intel PML4 ⇨Linux Page Global Directory (PGD)
  - Intel Page Directory Pointer Table ⇨Linux Page Upper Directory (PUD)
  - Intel Page Directory ⇨Linux Page Middle Directory (PMD)
  - Intel Page Table ⇨Linux Page Table (PTE)



Source: https://lwn.net/Articles/717293/

Key constants



And then there was 5!

- But 4 level tables still has lots of unused space
- A new level, P4D, is inserted between the PGD and the PUD
- Merged since kernel version 4.11-rc2
- But to date, there is no physical hardware support for 5 level paging
- Systems running with five-level paging will support 52-bit physical addresses and 57-bit virtual addresses

Canonical Addresses

- “No processors supporting the Intel 64 architecture support more than 48 physical-address bits.”
- The 16-bits (bits 63-48) of any 64-bit address must be a “sign extension” of Bit 47.
- Such an address is known as a canonical address.
- Intel manual says: “If a linear-memory reference is not in canonical form, the implementation should generate an exception.”

Physical Page Layout

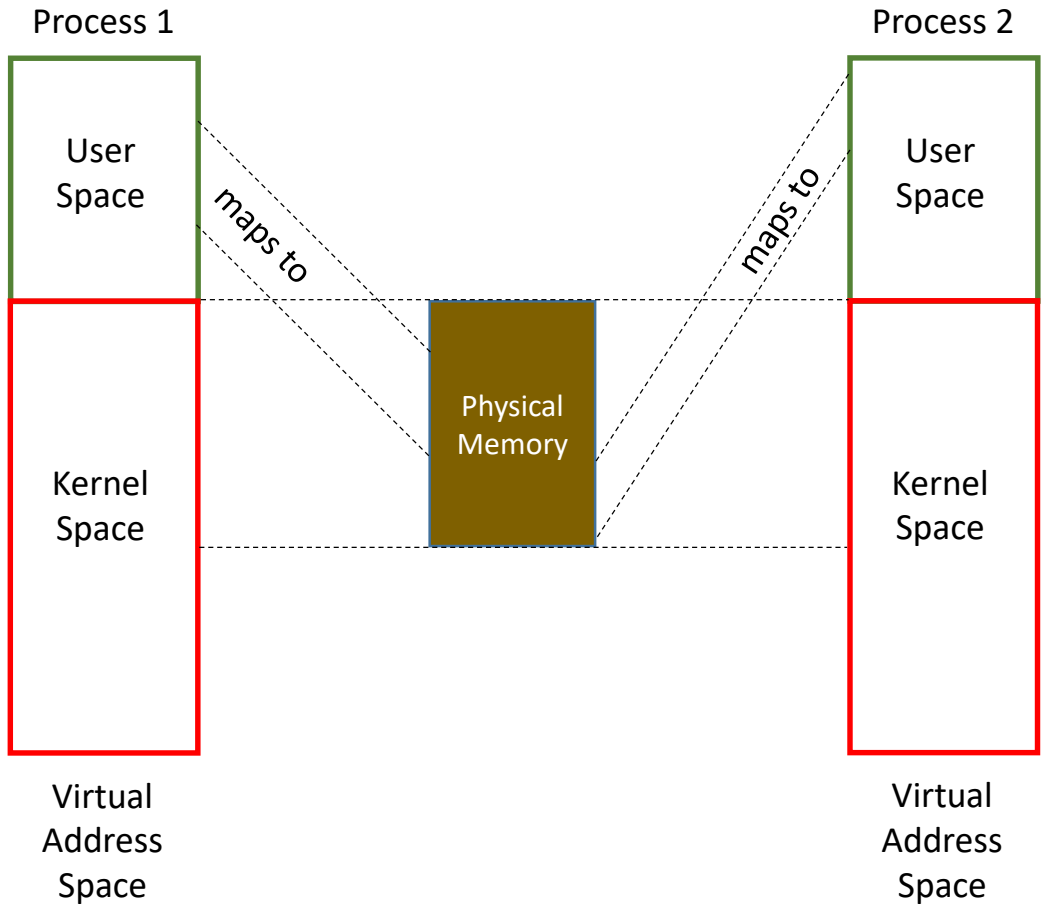
- During initialization, kernel builds a physical address map
- It identifies the reserved page frames that includes
  - Frames that are outside the available physical address range
  - Frames containing kernel code and initialized data structures
- A page contained in a reserved page frame can never be dynamically assigned or swapped to disk.
- Generally, Linux kernel is in RAM starting from physical address `0x00100000`
  - Gives deference to the BIOS

The 64-bit Linux Memory Map

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffbffffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffef00000000 - ffffffffffffffff (=64 GB) EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffff9fffffff (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - fffffffffff5fffff (=1526 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

View from 30,000 feet

- Some parts of user space of any process maps to parts of physical memory
- ALL of physical memory maps to a fixed region in kernel space
- Some parts of kernel space will map into parts of physical memory (vmalloc/ioremap)



Master Kernel Page Tables

- The kernel uses a single set of page tables
  - Never directly used but changes need to be propagated to all processes
    - Sharing at PGD level
  - Pointed to by the root variable **`swapper_pg_dir`**
  - Set up during initialization
  - Held in a master memory descriptor (**`init_mm`**)
  - It is the page tables of Process 0

- A copy of the kernel page tables is in every process
  - Updating it is very expensive – but only needed when a new kernel PGD entry is added
  - Uses a “deferred” approach by reference to the master copy

Direct mapping

- All of the available physical memory is mapped into the kernel virtual space as a contiguous block starting from **`__PAGE_OFFSET`**
  - Gives kernel full and easy access to any page frame
  - Protection has to be in place
  - For 32-bit, can be configured; normally **`__PAGE_OFFSET`** is **`0xC0000000`**
  - For 64-bit, **`__PAGE_OFFSET`** is **`0xffff880000000000`**

VMALLOC

- Kernel can allocate, deallocate non-contiguous memory using the `vmalloc()` and `vfree()` calls

- Works in a way similar to user level `malloc()` and `free()`

`fixmap` addresses

- A fix-mapped linear address is a constant linear address whose corresponding physical address does not have to be the linear address minus `__PAGE_OFFSET`, but rather a physical address set in an arbitrary way

- Used for supporting special hardware like APIC

- Also used for `vsyscall` implementation

Kernel and the Cache

- The most frequently used fields of a data structure are placed at the low offset within the data structure, so they can be cached in the same line.

- When allocating a large set of data structures, the kernel tries to store each of them in memory in such a way that all cache lines are used uniformly.

Working with the TLB

- It is the kernel, and *not* the hardware, that decides when a mapping between a linear and a physical address is no longer valid.

- Entire TLB may be flushed by certain changes
  - Such as changing CR3

- Or individual TLB entry
  - '`invlpg m`' x86 instruction will invalidate the TLB entry containing the address `m`.

Avoiding TLB flushes

TLB flush not needed if:

- The next process selected for scheduling has the same set of page tables as the current one

- Scheduler selects a kernel thread
  - It “uses” the same page tables as the current process
    - Kernel space is largely the same for all processes

Page Descriptor

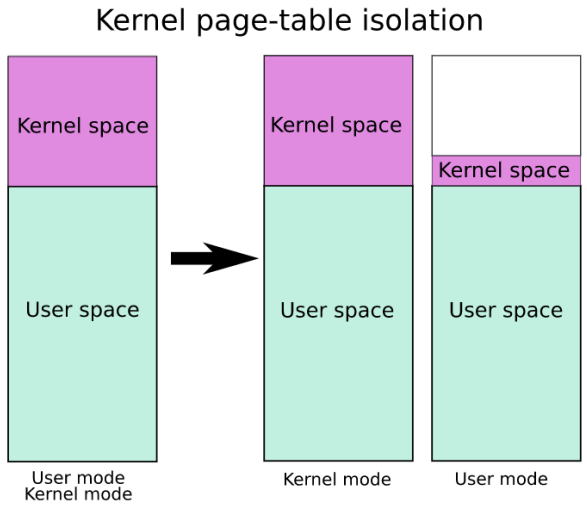
- "`struct page`" in include/linux/mm_types.h

- One per page frame
  - 32 bytes or more
  - Keeps the key information about the page frame including permissions, reference counts etc.

Page Table Isolation

Page Table Isolation (PTI)

- Previously known as KAISER
- Countermeasure against attacks on the shared user/kernel address space
- Basic idea:
  - Two sets of page tables
  - When the kernel is entered via syscalls, interrupts or exceptions, the page tables are switched to the full “kernel” copy
  - When the system switches back to user mode, the user copy is used again
  - The userspace page tables contain only a minimal amount of kernel data



The full set of page tables

- A complete mapping of userspace that the kernel can use for things like copy_to_user()

- The user portion of the kernel page tables is crippled by setting the "not executable" (NX) bit in the top level
  - Any mistake in switching CR3 to userspace will cause an exception

The userspace page mapping

- Userspace page tables map only the kernel data needed to enter and exit the kernel
  - This data is entirely contained in the `struct cpu_entry_area` structure

- Ensures in userspace cannot access the bulk of kernel mappings

Shadowed CR3

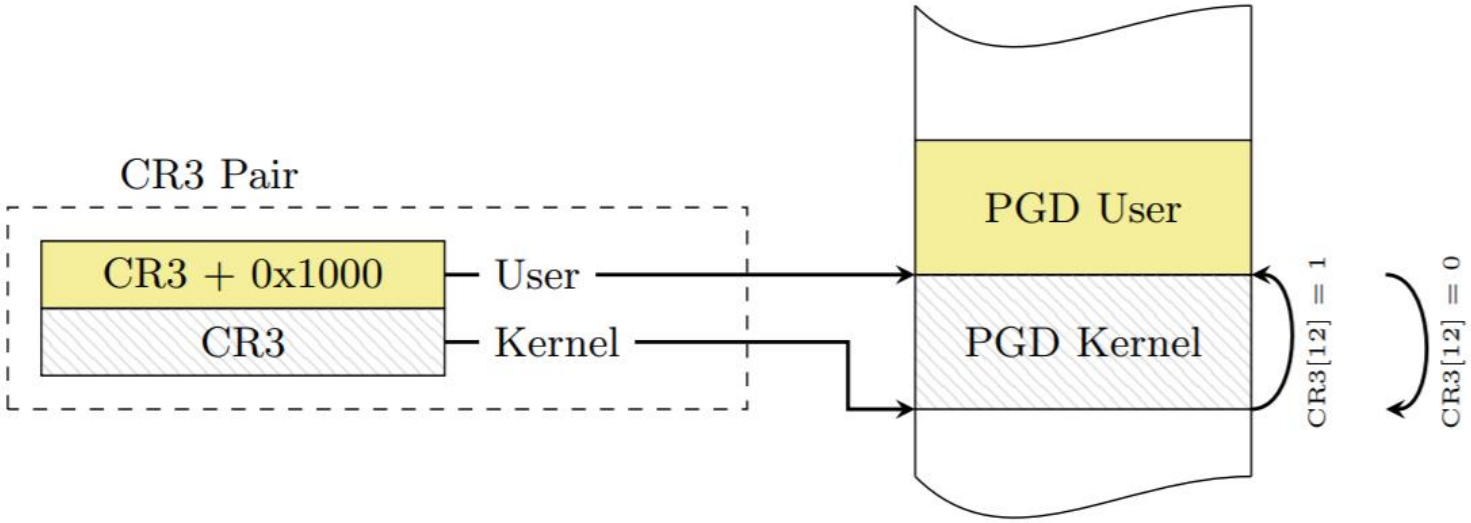


Fig. 3: Shadow address space: PML4 of user address space and kernel address space are placed next to each other in physical memory. This allows to switch between both mappings by applying a bit mask to the `CR3` register.

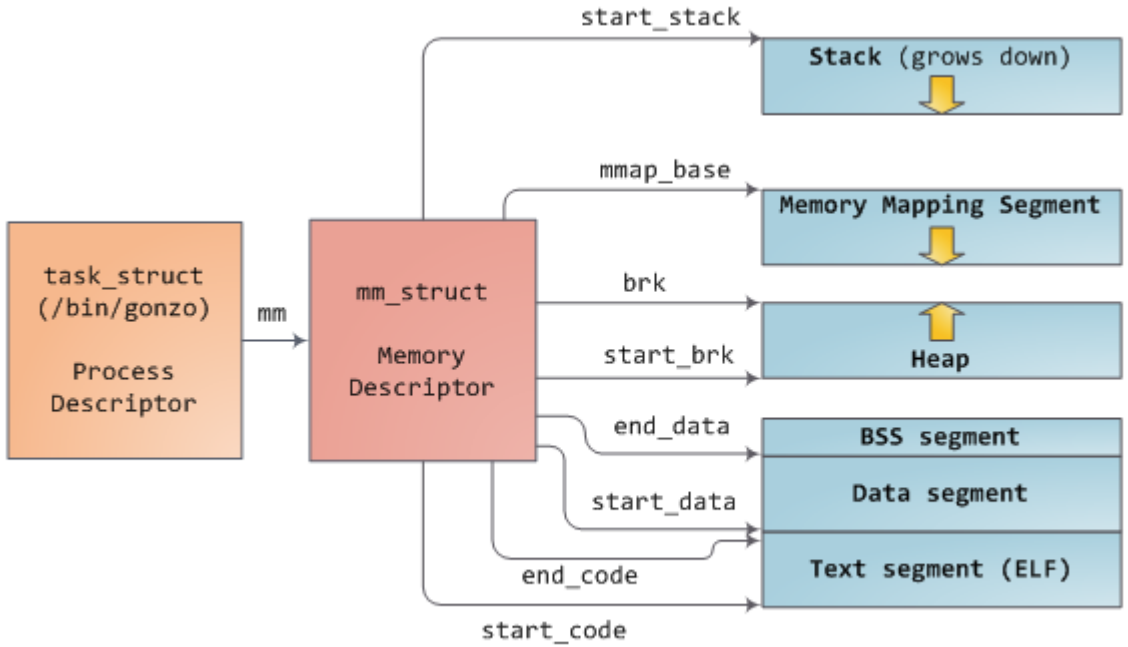
Source: https://gruss.cc/files/kaiser.pdf

Linux Memory Management

Virtual memory of a process

- First set up by the loader using the ELF of the application
- Heap area virtual pages are allocated using **`brk()`**
- Alternatively, files can be **`mmap()`** into the virtual address space

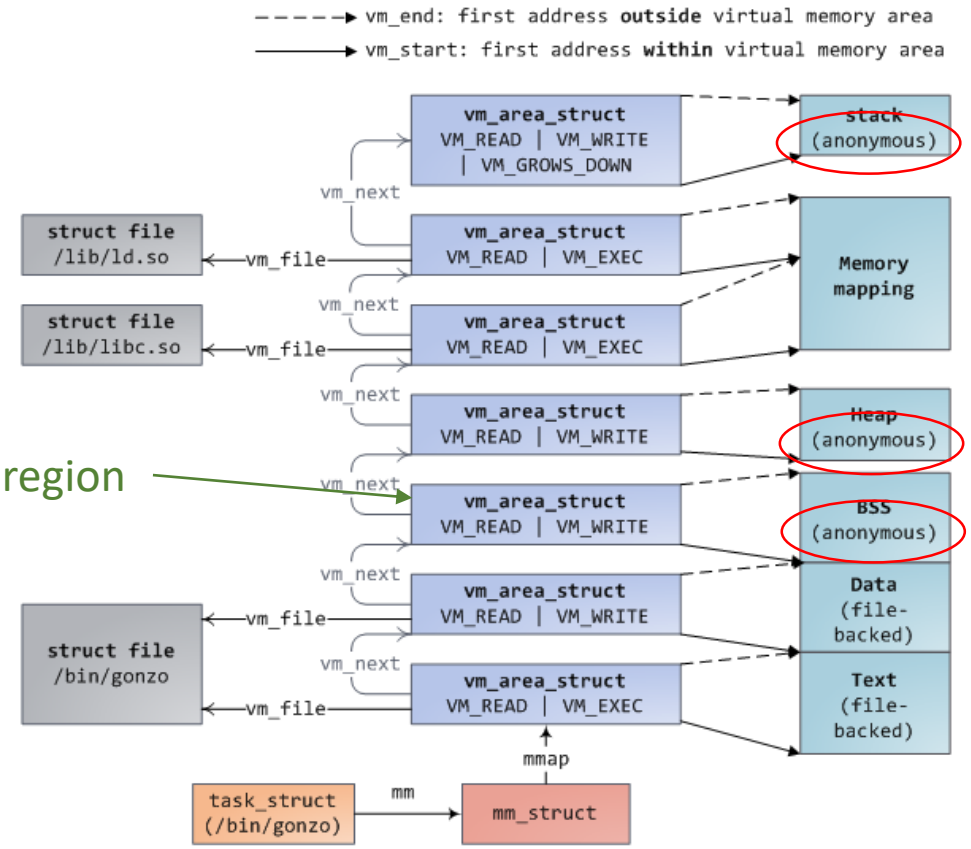
User Level Processes



Source: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/

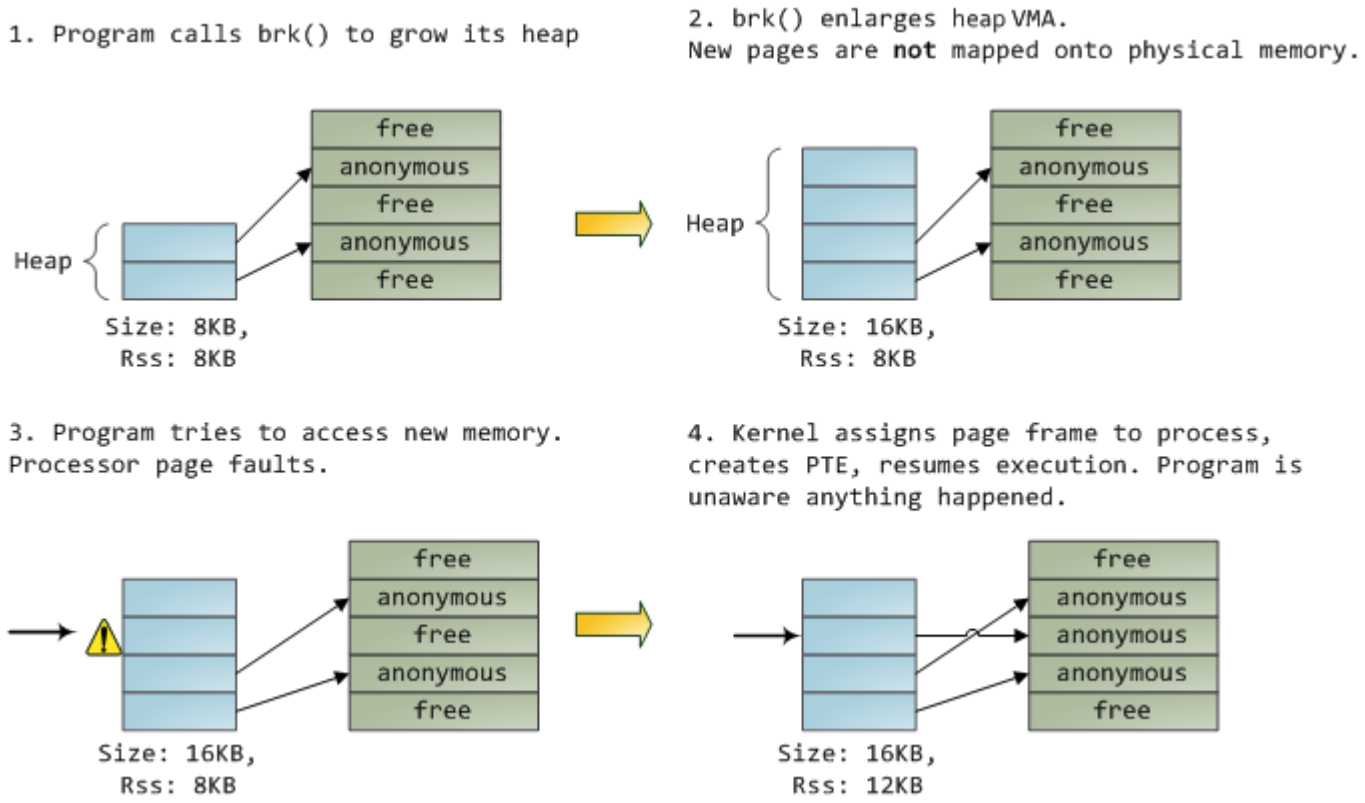
Virtual Memory Area

Also called a (virtual) memory region



Source: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/

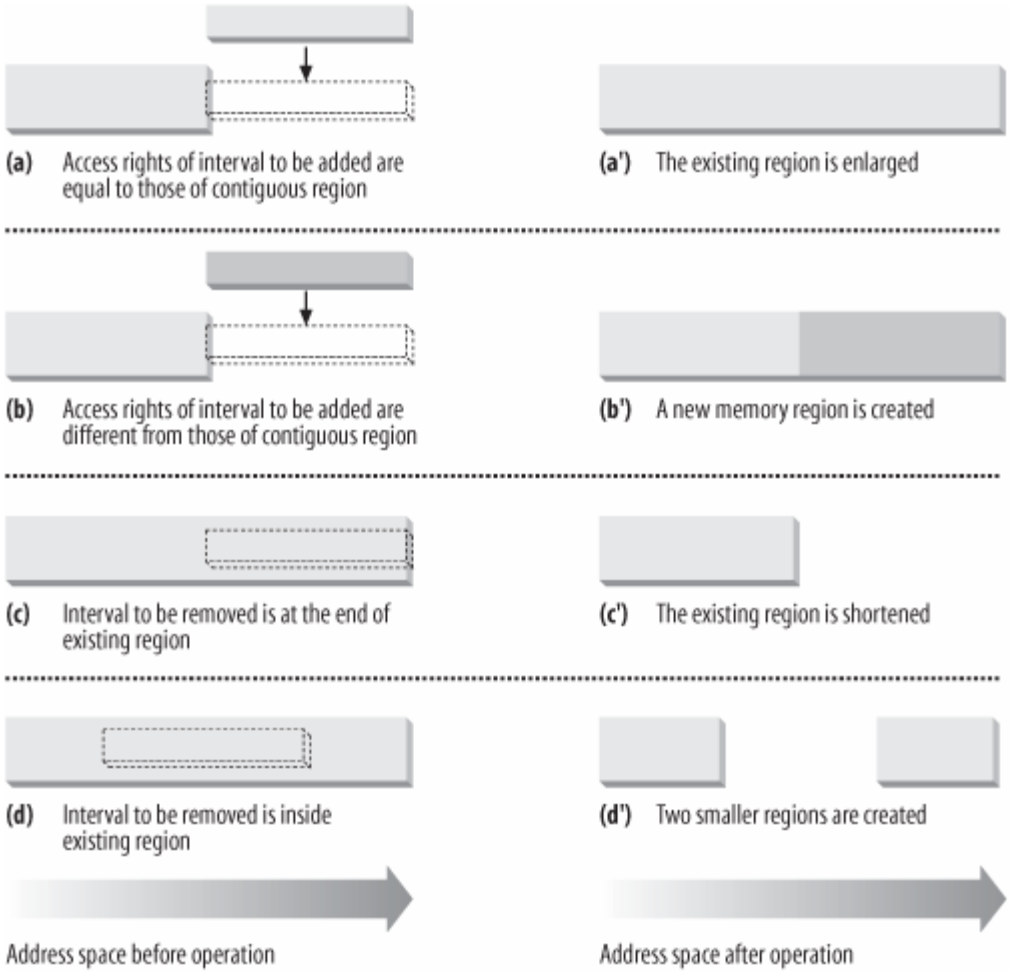
Lazy Expansion



Source: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/

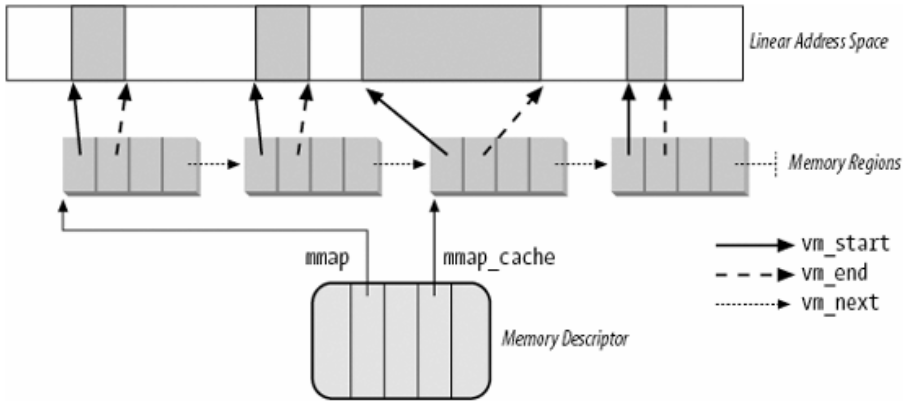
VMA operations

- When a new file is mmap'ed
- When a new shared memory segment is created



Searching the VMA

- A frequent operation is to find the region a virtual address is in
- Either using the linked list to search linearly
- Or (for processes with many memory regions) - a red-black tree



On Physical Memory

Memory Zone

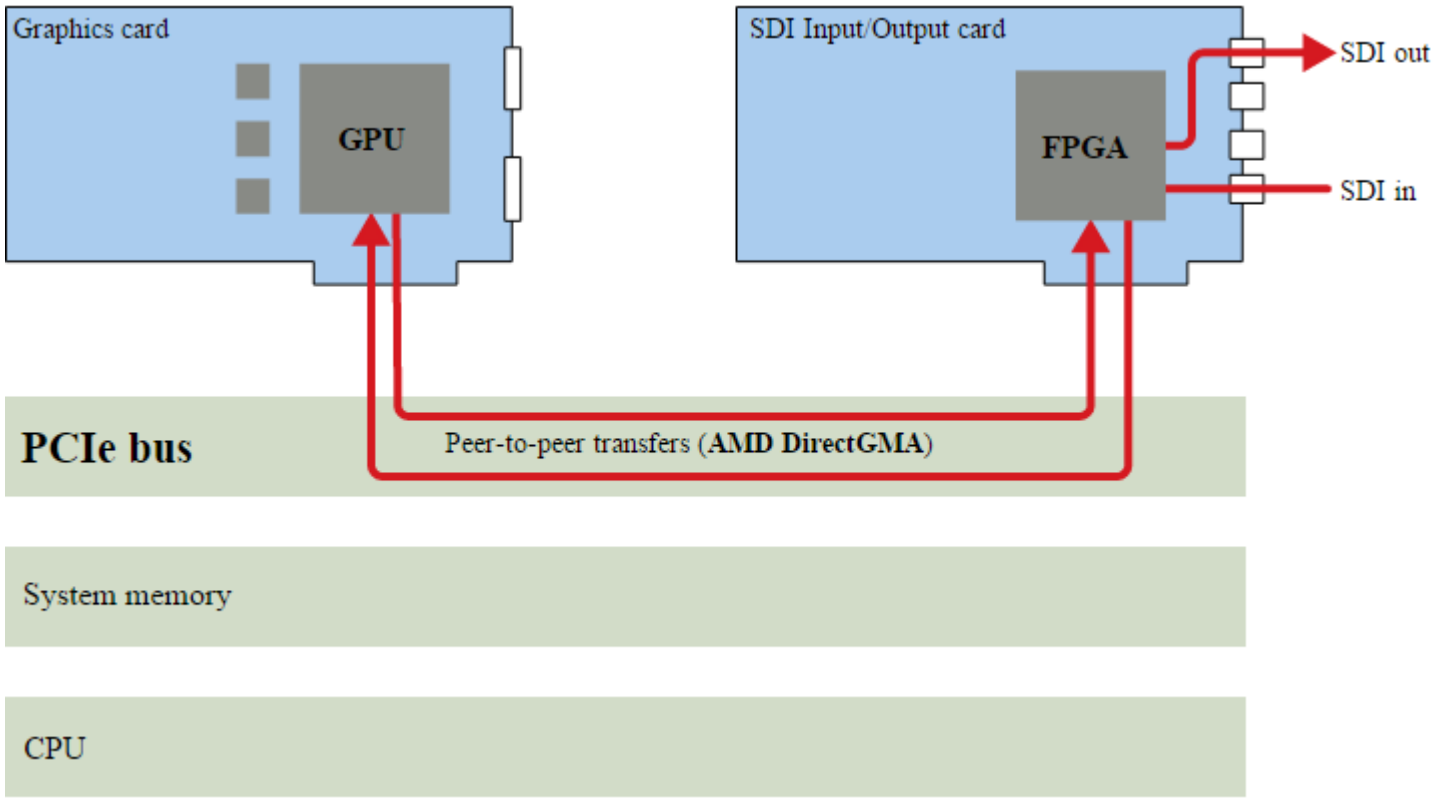
- Physical memory is divided into zones

- One major reason: direct-memory access
  - Devices read/write data directly into memory without CPU involvement after setup

Direct Memory Access

- Hardware subsystem direct access to memory independent of CPU
  - Frees up the CPU
- Can also be used for memory to memory copying, or scatter-gather operations
- Used by many including disk drive controllers, graphics cards, network cards and sound cards.

Example: AMD DirectGMA



DMA modes of operations

- Burst mode / Block transfer mode
  - DMA controls bus and fastest in transfer
- Cycle stealing mode
  - Interleave bus with CPU
- Transparent mode
  - Slowest mode: DMA uses bus only when CPU is not

The trouble with DMA

- DMA works only with physical addresses
  - OS must set it up
- Legacy DMA have limited address bits
  - Legacy x86 only up to 16MB of physical memory
- PCI-e can do 32 or 64 bit DMA
  - 64-bit almost never implemented/used

Memory Zone

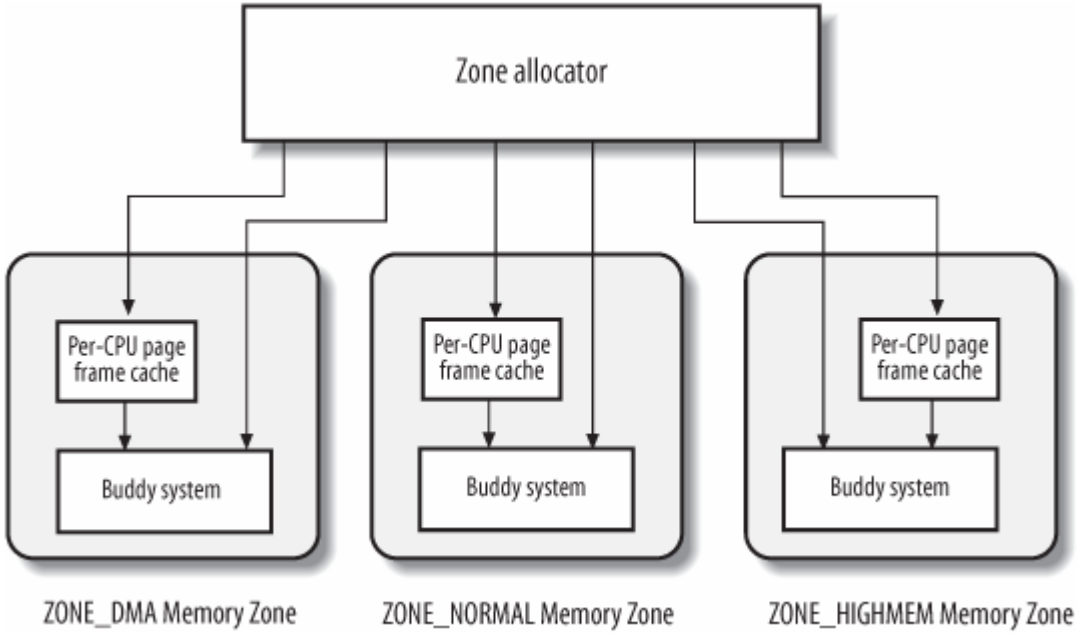
- Part of physical memory must be set aside for DMA use
  - And they must be in certain ranges
  - In x86-64, **`ZONE_DMA`** (16MB) and **`ZONE_DMA32`** (4GB)

- Others: **`ZONE_NORMAL`**, **`ZONE_HIGHMEM`**, **`ZONE_DEVICE`**

- Page allocation must be done to the correct zone

The Zoned Page Frame Allocator

- Zone allocator receives requests for allocation and deallocation of dynamic memory
- Actual work done by the Buddy System Algorithm
- For better performance, a small cache is used for fast, single frame allocation requests



Reserved Page Frame Pool

- In normal memory allocation, if not enough free memory, must put request on hold then do memory reclamation
  - But some request must not block in the kernel (`GFP_ATOMIC`), e.g. inside interrupt servicing routines
- Solution: reserve a small amount of free memory for atomic allocation
- Need to “top up” if too low

Different needs

- **`kmalloc()`** and **`kfree()`**
  - Allocates contiguous physical memory
  - A “must” for DMA especially
- **`vmalloc()`** and **`vfree()`**
  - Allocates contiguous virtual memory
- Strictly for use inside the kernel

External Fragmentation

- Allocation and deallocation of groups of contiguous page frames of different sizes may lead to a situation of several small blocks of free page frames scattered inside blocks of allocated page frames

- Linux uses the Buddy Allocation Algorithm

The Buddy System Algorithm

- Used by the kernel for allocating groups of contiguous page frames. (**`mm/page_alloc.c`**)
- All free page frames are grouped into 11 lists of blocks that contain groups of 1 (2^0), 2 (2^1), 4 (2^2), 8 (2^3), 16 (2^4), 32 (2^5), 64 (2^6), 128 (2^7), 256 (2^8), 512 (2^9), and 1024 (2^{10}) contiguous page frames
- The physical address of the first page frame of a block is a multiple of the group size.
  - E.g. the initial address of a 16-page-frame block is a multiple of 16 x 4096 (4K page)

Allocating memory via the Buddy System

- The algorithm for allocating, for example, a block of 256 contiguous page frames
  - First checks for a free block in the 256-page-frame list
  - If no free block, it then looks in the 512-page-frame list for a free block
  - If it finds a block, the kernel allocates 256 of the 512 page frames and puts the remaining 256 into the list of free 256-page-frame blocks.
  - If no free 512-page block, kernel looks at the next larger list (i.e., a free 1024-page-frame block) allocating it and dividing the block similarly
  - If no block can be allocated an error is reported

Reclaiming memory in the Buddy System

- The kernel attempts to merge pairs of free buddy blocks of size *b* together into a single block of size *2b*. Two blocks are considered buddies if:
  - Both have the same size.
  - They are located in contiguous physical addresses.
  - The physical address of the first page frame of the first block is a multiple of 2 *x b x* 2^{12}.
- This is an iterative algorithm; if it successfully merges released blocks, *b* is doubled and bigger blocks are attempted.

Buddy System Data Structures

Linux 2.6 uses a different buddy system for each zone. Each one relies on the following main data structures:

- **`mem_map`** array – contains all the page frame descriptors on the system
- An array having 11 elements of type **`free_area`**, one element for each group size.

Interfaces

- Buddy algorithm triggered in **`alloc_pages()`** and **`free_pages()`**
- Real work done in **`__rmqueue()`** and **`__free_pages_bulk()`** in **`page_alloc.c`**
- “GFP” – “get free pages”

/proc/buddyinfo

- Used primarily for diagnosing memory fragmentation issues. Using the buddy algorithm, each column represents the number of pages of a certain order (a certain size) that are available at any given time.
- The DMA row references the first 16 MB on a system, the HighMem row references all memory greater than 4 GB on a system, and the Normal row references all memory in between.

```
Node 0, zone      DMA  90    6     2     1     1     …
Node 0, zone   Normal  1650  310   5     0     0     …
Node 0, zone  HighMem  2     0     0     1     1     …
```

Per-CPU Page Frame Cache

- For each CPU, pre-allocate some page frames to be used by that CPU

- Hot cache
  - Good if kernel or user mode process running on that CPU will write immediately to the frame

- Cold cache
  - For DMA operation

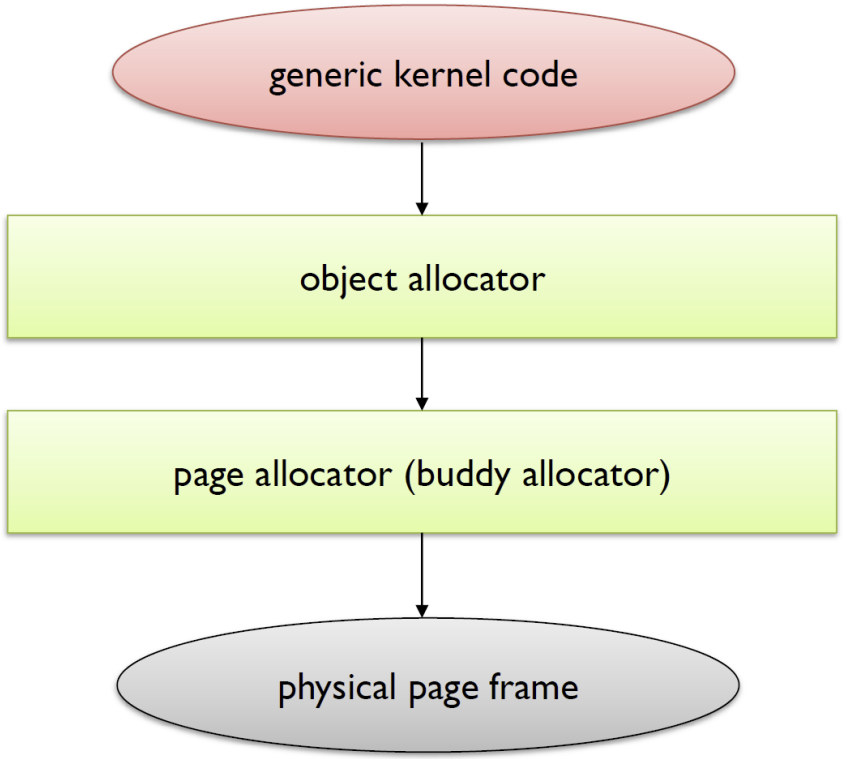
- Interface: **`buffered_rmqueue()`**

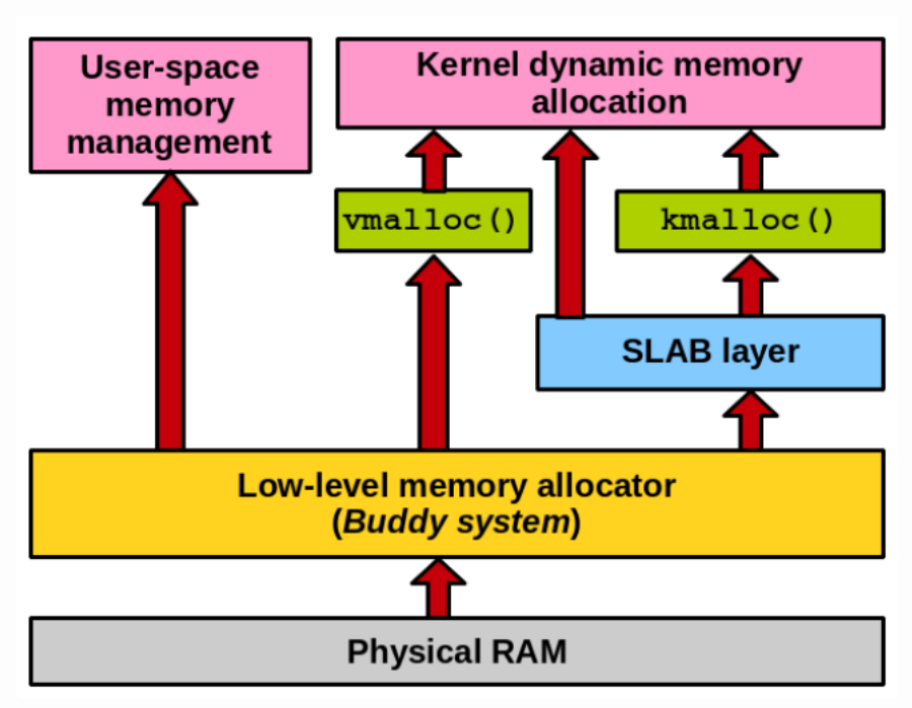
Internal Fragmentation

- Buddy algorithm eliminates external fragmentation (between pages)

- But internal fragmentation still an issue
  - Objects smaller than a page
  - Objects of different sizes are allocated and freed, resulting in “holes” inside a page

The memory allocators





vmalloc() vs kmalloc()

- Both are memory allocators for the kernel
- **`kmalloc()`**
  - Allocated pages are physically as well as virtually contiguous
  - GFP flags
    - **`GFP_ATOMIC`** – atomic (either succeed or fail), never sleeps, highest priority
    - **`GFP_KERNEL`** – "normal", allowed to sleep
- **`vmalloc()`**
  - Allocated pages are virtually contiguous but not necessarily physical contiguous
  - The kernel's **`malloc()`**
  - Will still work if there is insufficient physical memory

Kernel Object Allocators

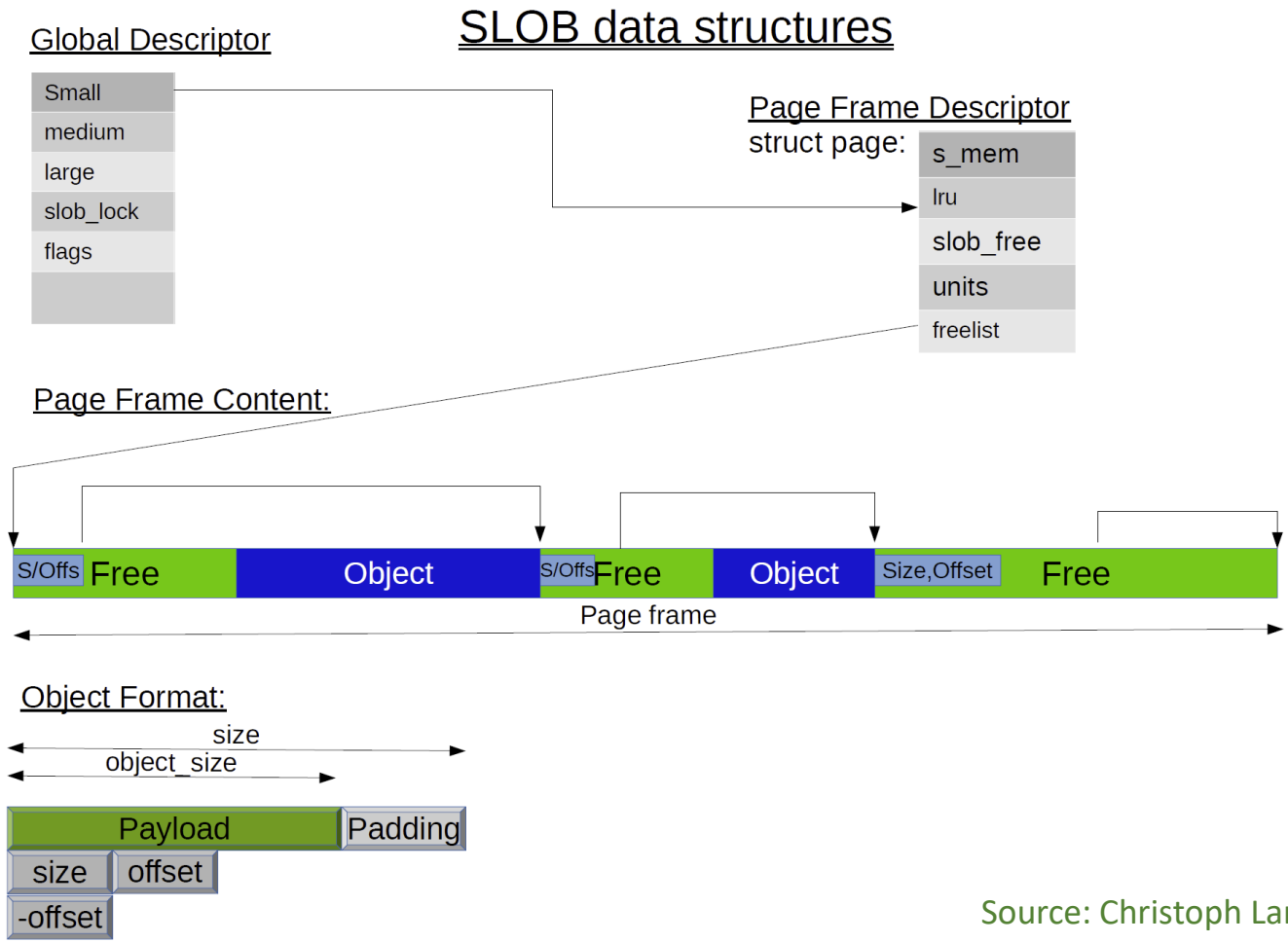
The Kernel Object Allocators

- SLOB (Simple List of Blocks) – K&R 1999 vintage
  - Goal: compactness of memory, minimum memory overhead
- SLAB – Pioneered by Solaris OS
  - Goal: cache friendly, waste some space to gain cache efficiency
- SLUB – current
  - Goal: speed

The SLOB Allocator

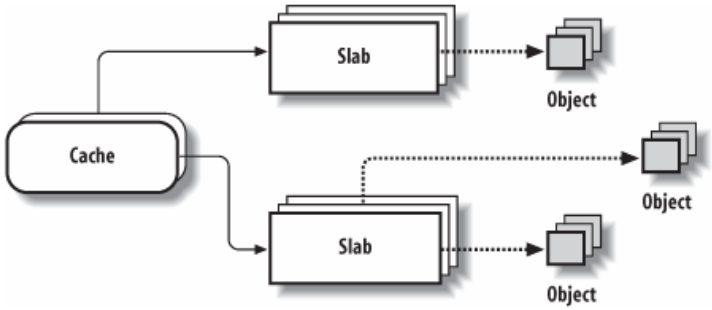
- First proposal: first-fit
  - Single list of allocated and free space
  - Object allocation: find first free space that is bigger than requested, cut it up and allocate to requester; if none available, enlarge the heap
  - Object deallocation: try to merge with neighbouring free space
- 1st optimization: use list of different object sizes
- 2nd optimization: best-fit
  - Search for the smallest in list that will satisfy request

The SLOB Allocator



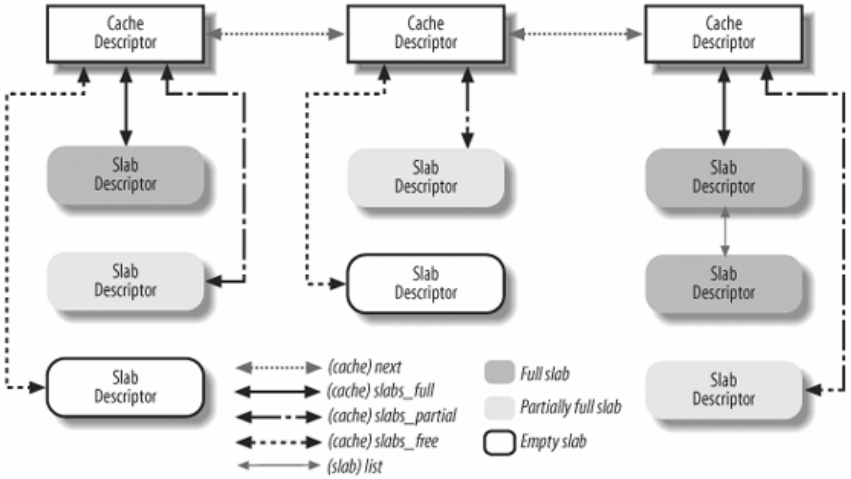
Source: Christoph Lameter, LinuxCon 2014

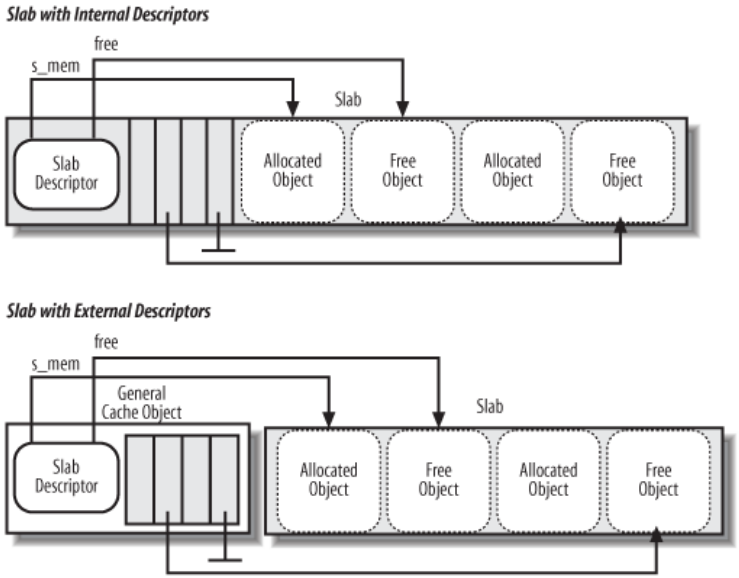
The SLAB Allocator



- Pioneered by Solaris OS
- Based on the idea of caches
- Each cache stores objects of the same type (hence size)
  - Insight: the OS works often with objects of particular types
- In general, 26 general cache descriptors associated with memory areas of size 32 bytes to 128Kbytes.
  - For each size, there are two caches: one suitable for ISA DMA allocations and the other for normal allocations.

The Descriptors





- The objects managed by the slab allocator are aligned in memory.
- If the object's size is greater than half of a cache line, it is aligned in RAM to a multiple of **`L1_CACHE_BYTES`**.
- Otherwise, the object size is rounded up to a submultiple of **`L1_CACHE_BYTES`**
  - Ensures that a small object will never span across two cache lines.

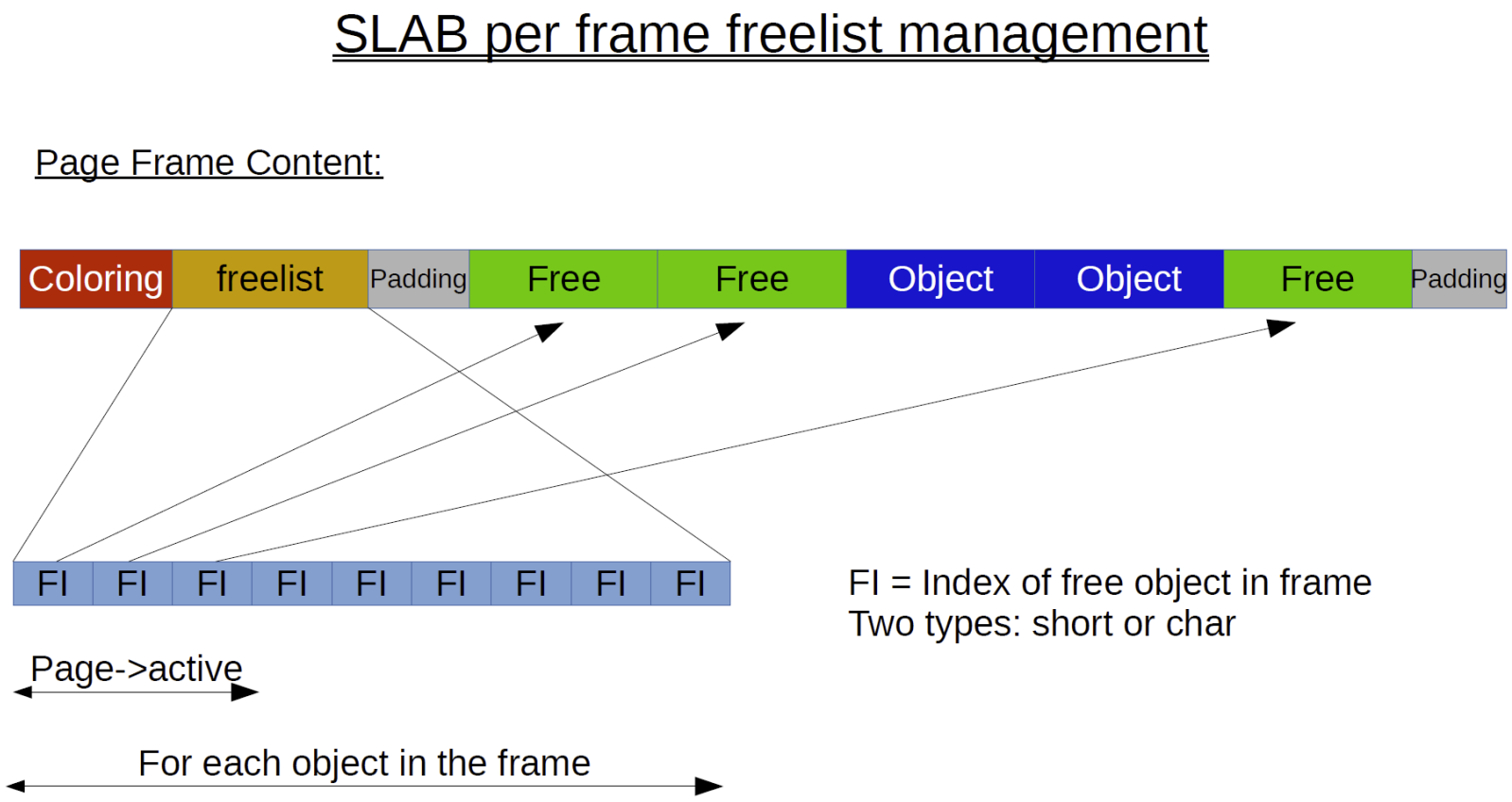
Allocating and deallocating slabs

- A slab consists of a few contiguous pages
- A slab is allocated when there is no free space available on an allocation request
  - Caches are empty when created until the first allocation request
- Slabs are destroyed when there are too many free objects

Slab coloring

- Objects that have the same offset within different slabs may (with high probability) land in the same cache line
  - Risk cache thrashing
- There will be some free space in a frame
  - Will be less than an object
- Idea: use a “colour” to offset the addresses in the frame
  - Different colour slabs will have different initial offset of the first object

The SLAB data structure

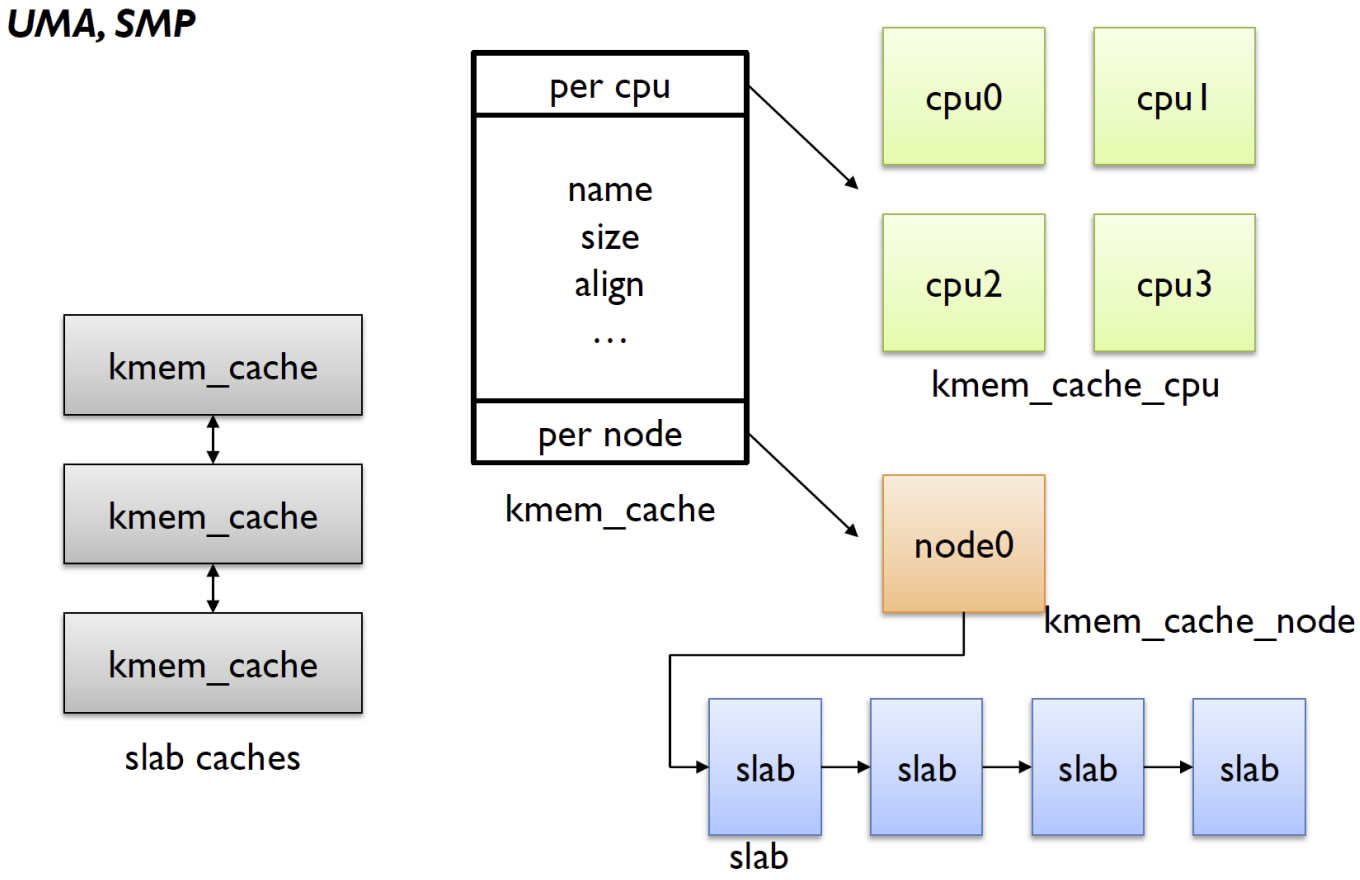


Source: Christoph Lameter, LinuxCon 2014

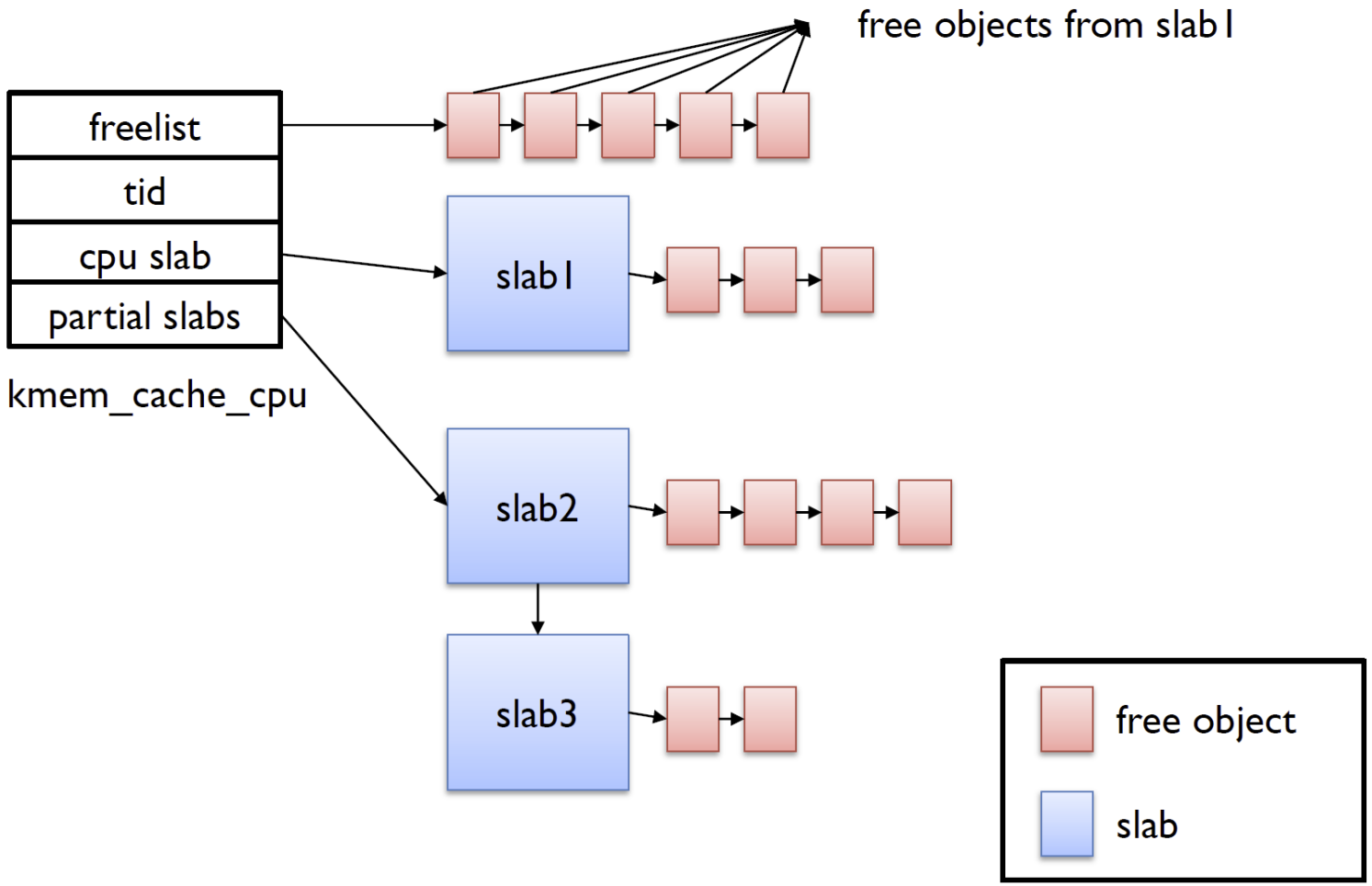
The SLUB Allocator

- Improved version of the SLAB allocator
- Improved support for debugging
  - Off by default

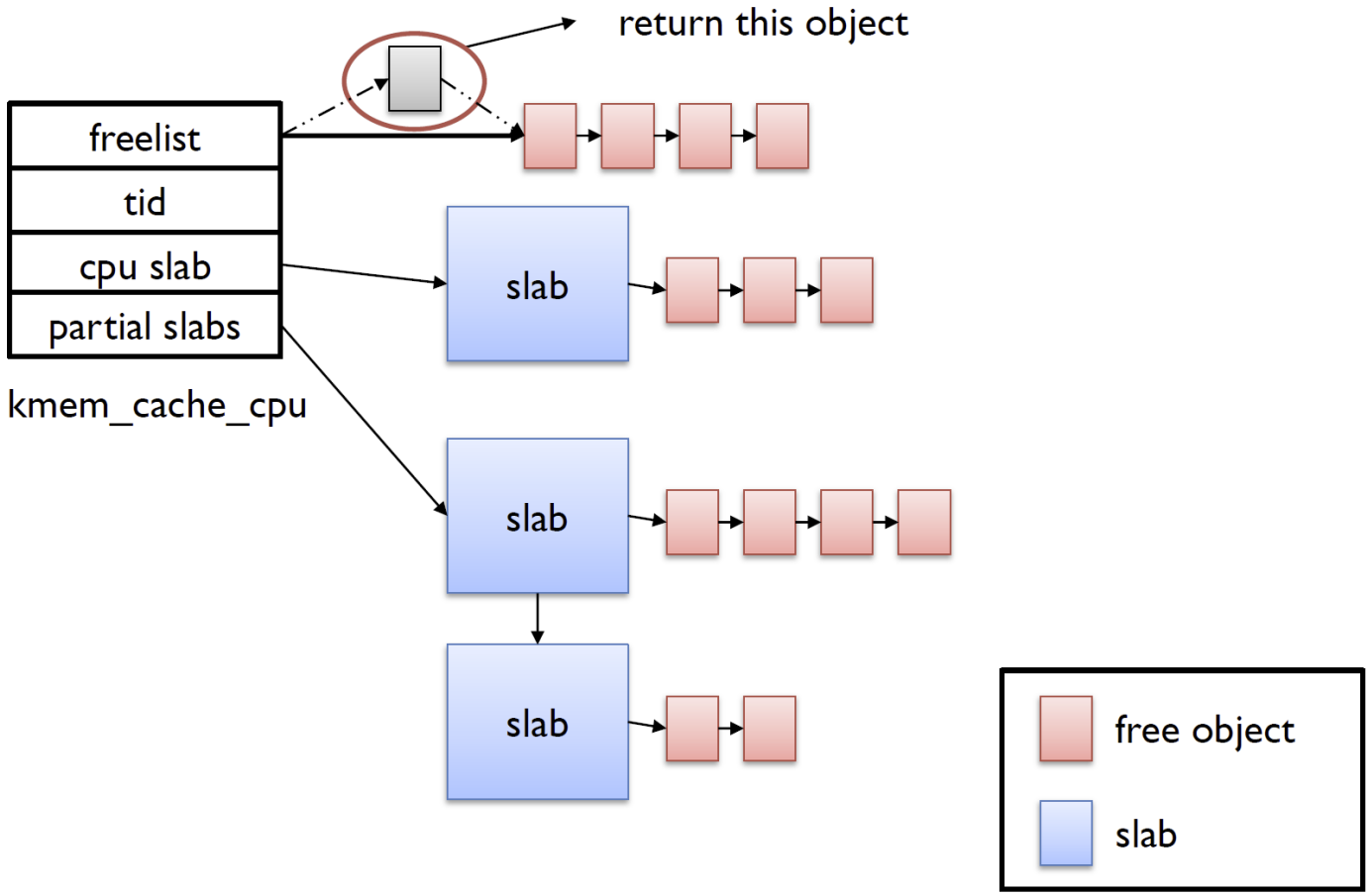
SLUB Data Structures



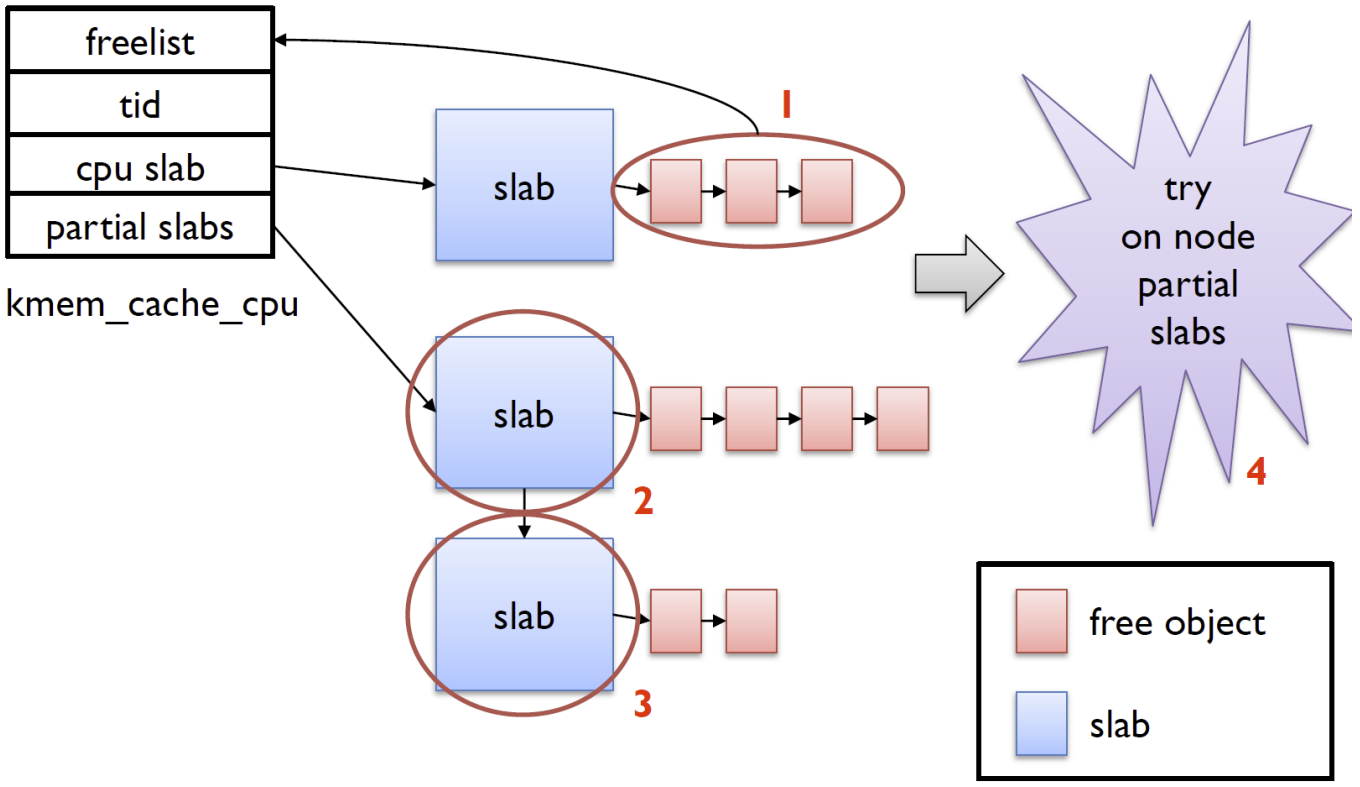
Per-CPU lists



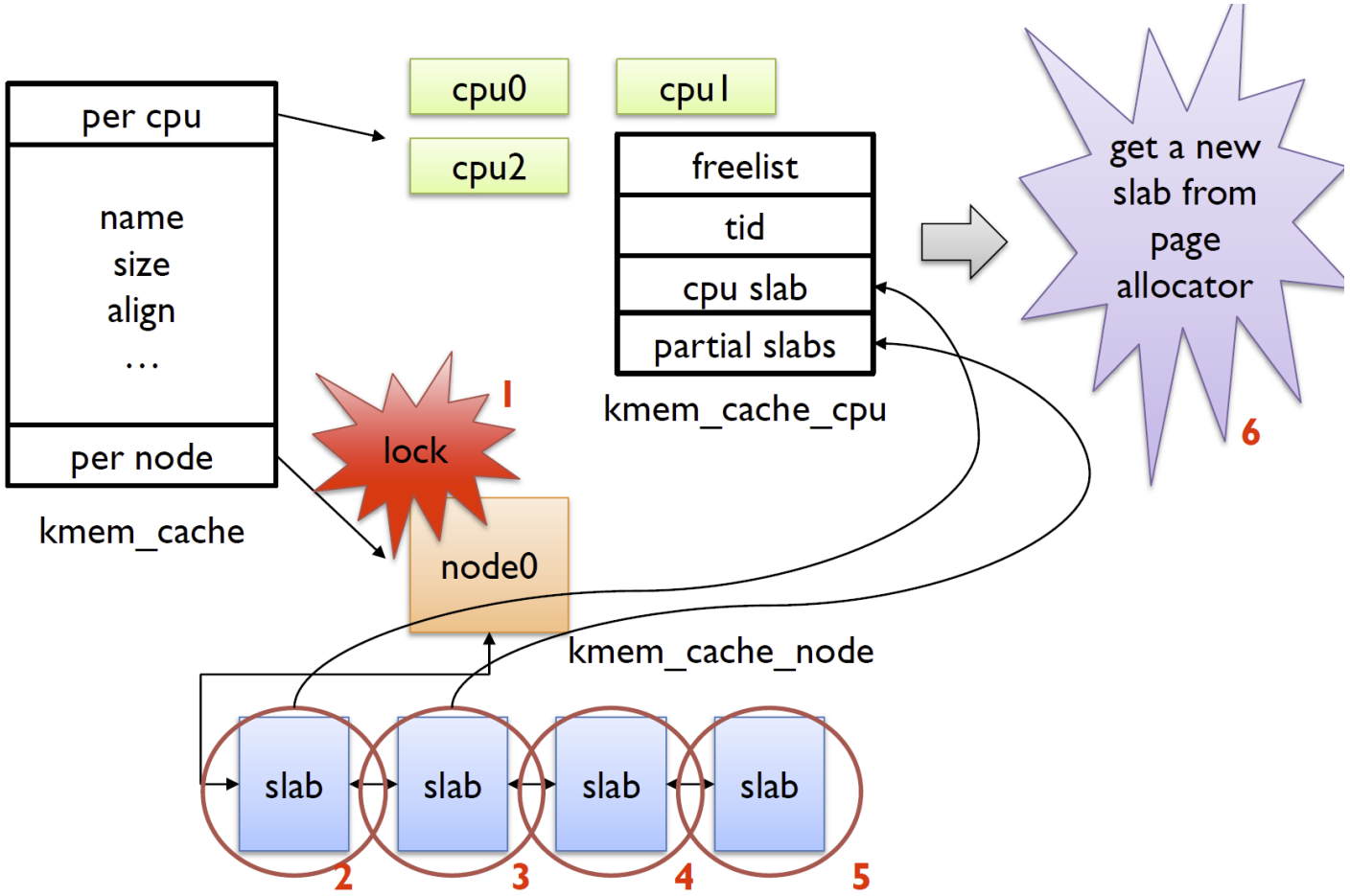
Allocation: Fast Path



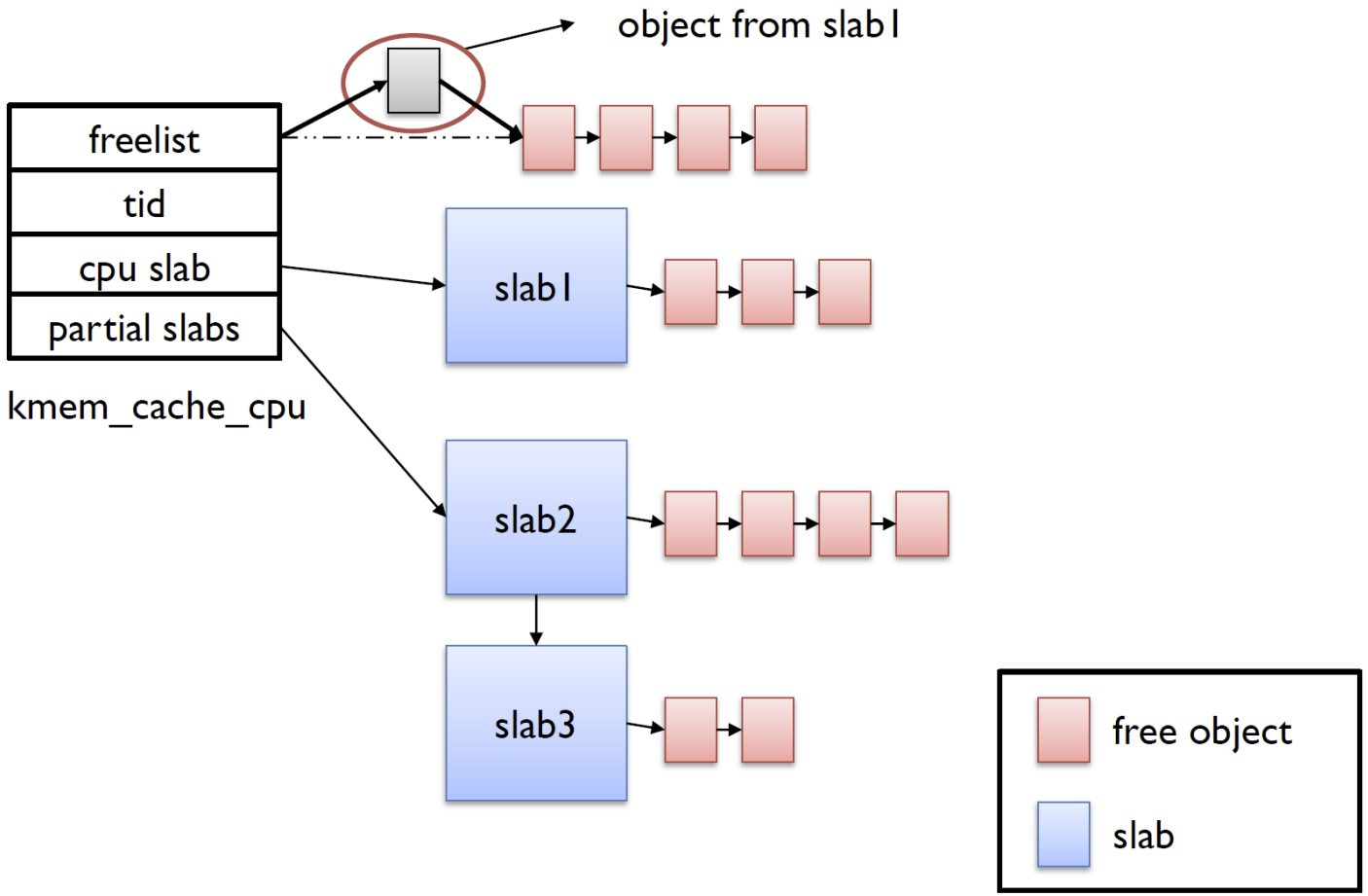
Allocation: Slow Path



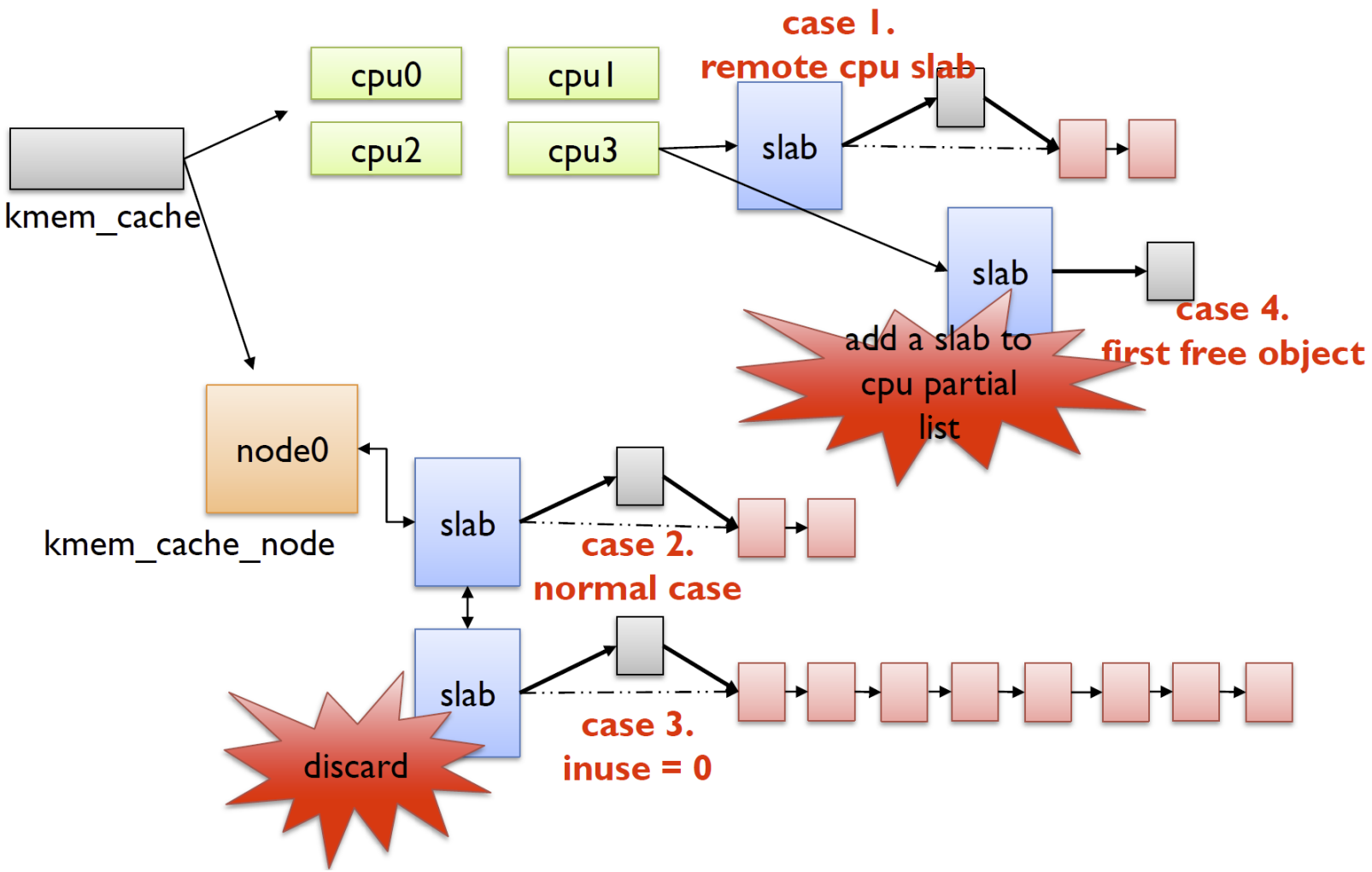
Allocation: Very Slow Path



Free: Fast Path

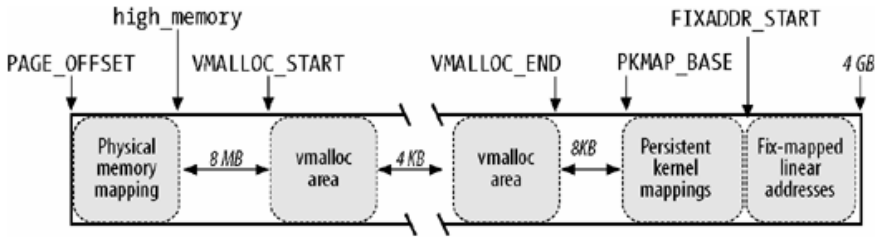


Free: Slow Path



Noncontiguous Memory

- For infrequent memory allocation requests with no hard requirement of being in contiguous page frames
  - Virtual (linear) address can still be contiguous
- Used for swap areas and modules
- Access via the **`vmalloc()`** interface



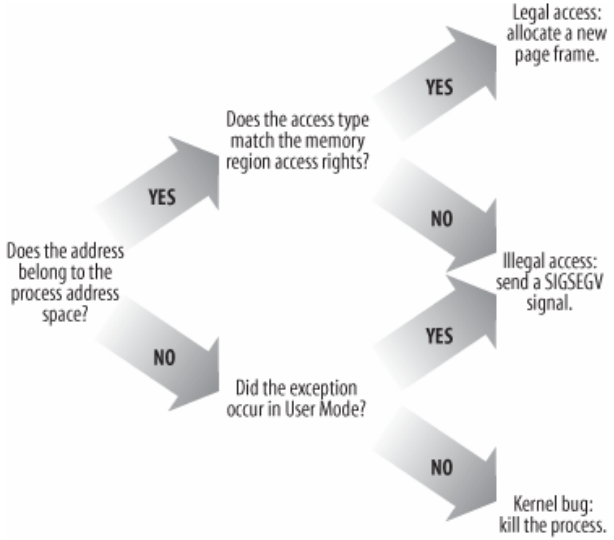
(for 32bit kernel)

The Page Fault Handler

Page fault

- Page size in Linux is 4KB

- On a page exception
  - Is the page already in a frame?
  - Does the process have permission to access it?
  - If page not in frame, must take further action.



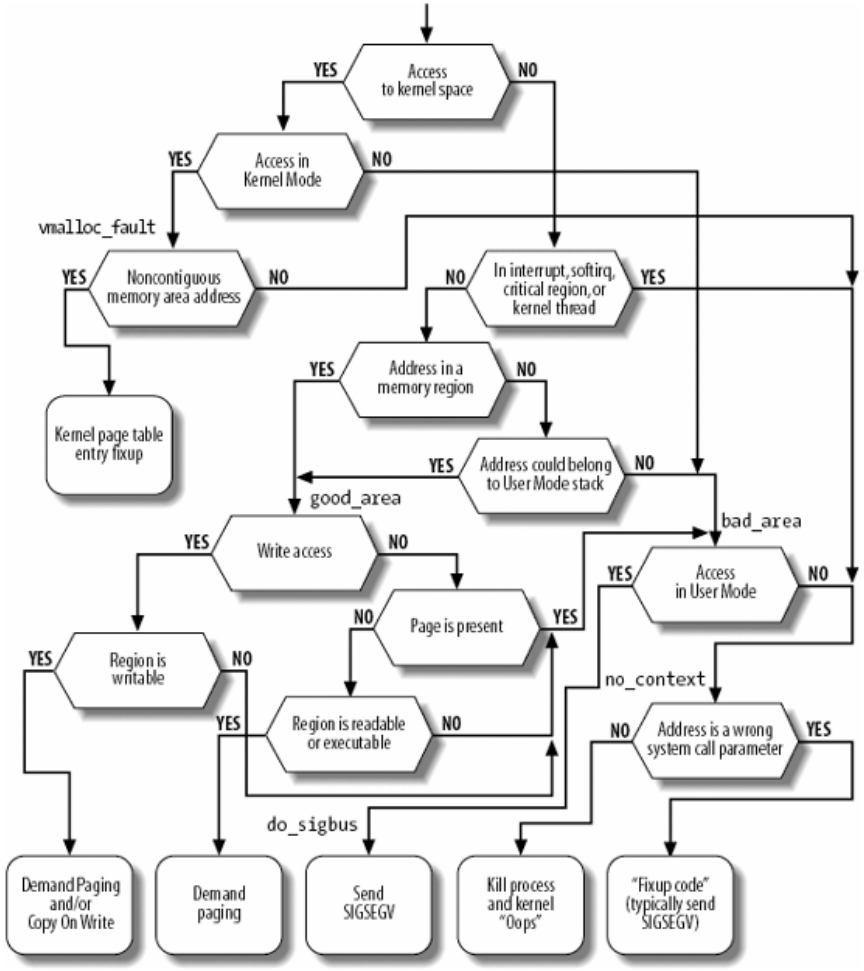
- The kernel itself can also page fault but only to handle copying to and from userspace.

- The Linux kernel itself is **NOT PAGEABLE**.
  - Kernel threads may be holding locks, or disabled interrupts

Page fault on x86

- Entry point **`do_page_fault()`** in **`arch/x86/mm/fault.c`**
- Linear address causing the fault is captured in **`CR2`**
- Minor fault: do not need to block the current process
- Major fault: must put current process to sleep
  - Demand paging
- **`VM_FAULT_OOM`**: Out of memory, cannot find a page frame, kill process
- Executes the flowchart

The flow diagram



Page Frame Reclamation and Swapping

The issues

- Page frames are precious resource
  - Even when allocated, they may not stay committed to the same process for its entirety
- Memory allocators (user and kernel) allocate and deallocate memory but page frames even if allocated may be “swapped out”
- Another complication: several page table entries may point to the same frame
  - If a frame is reclaimed, and a PTE is not updated correctly, OS is incorrect
  - How to find it?

Goal of PFRA

- To ensure system is functioning with a healthy availability of free page frames
- Unfortunately, different workload imposes different needs
- Empirical heuristics used in its design
  - Hard to formulate a theoretical framework

Page Frame Reclamation Algorithm (PFRA)

- The core algorithm for “stealing” page frames to increase the free list of the buddy system

- It is performed before all free frames are used up
  - Otherwise, could crash the system
  - PFRA itself need to use frames to work

The Four Types of Page Frames

Type of pages	Description	Reclaim action
Unreclaimable	Free pages (included in buddy system lists) Reserved pages (with `PG_reserved` flag set) Pages dynamically allocated by the kernel Pages in the Kernel Mode stacks of the processes Temporarily locked pages (with `PG_locked` flag set) Memory locked pages (in memory regions with `VM_LOCKED` flag set)	(No reclaiming allowed or needed)
Swappable	Anonymous pages in User Mode address spaces Mapped pages of *tmpfs* filesystem (e.g., pages of IPC shared memory)	Save the page contents in a swap area
Syncable	Mapped pages in User Mode address spaces Pages included in the page cache and containing data of disk files Block device buffer pages Pages of some disk caches (e.g., the inode cache)	Synchronize the page with its image on disk, if necessary
Discardable	Unused pages included in memory caches (e.g., slab allocator caches) Unused pages of the dentry cache	Nothing to be done

Source: Bovet, Daniel P. Understanding the Linux Kernel.

Mapped vs Anonymous

- A page is said to be mapped if it is part of a file
  - To reclaim a mapped page, check if it is dirty (i.e., has been written to) and write the content to the corresponding file

- A page is anonymous if it is not mapped
  - Content must be saved to a dedicated disk partition or disk file called the swap area

Shared vs Non-shared page frames

- A page frame is shared if it belongs to more than one user process
  - Note: the same shared page frame may be mapped to altogether different virtual addresses in different processes – or even in the same process
- Otherwise, it is non-shared

General principles of Linux PFRA

- Free “harmless” pages first
  - Examples: infrequently used pages of various caches in the kernel
- Make all user mode pages reclaimable
- Reclaim a shared page frame by unmapping all PTEs referencing it in a single go
- Reclaim “unused” pages only

Reverse Mapping

Reverse Mapping

- Aim: to quickly locate all page table entries point to the same page frame

- Trivial solution: keep a reverse pointer in each page descriptor to chain up the PTEs
  - Keeping such a list up to date requires too much work

Object-based Reverse Mapping

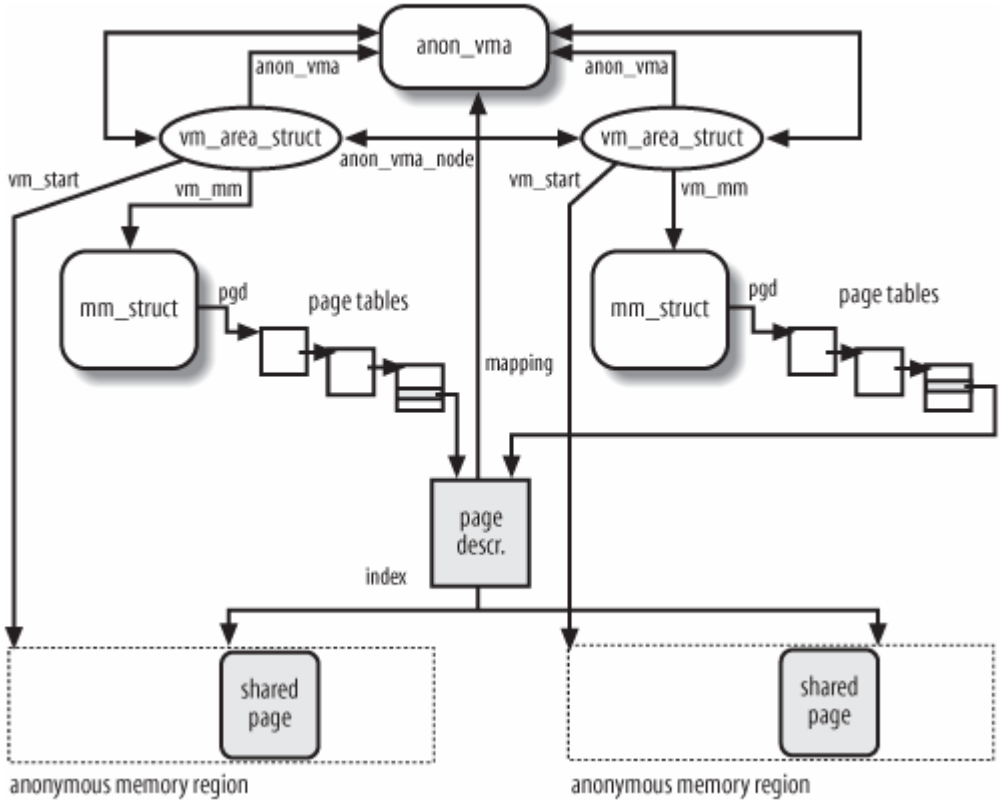
- Introduced in Linux 2.6
- Recall: every page frame has a corresponding page descriptor
- Page descriptor has two fields: `_mapcount` and `mapping`
- `_mapcount`
  - -1 : no PTE points to this page
  - 0 : page is non-shared, only one PTE points to it
  - > 0 : page is shared, `_mapcount` number of PTEs point to it
  - `page_mapcount()` returns `_mapcount` + 1

`mapping` field

- If null, the page belongs to the swap cache
  - A cache for moving pages to and from swap disk
- If non-null, and LSB is 1
  - Page is anonymous
  - Mapping field is points to an `anon_vma` descriptor
- If non-null, and LSB is 0
  - Page is mapped
  - Mapping field points to the `address_space` object of the page cache

Reverse Mapping for Anonymous Pages

A doubly linked list collects all the memory regions that include the same page frame



Source: Bovet, Daniel P. Understanding the Linux Kernel.

Reverse Mapping for Anonymous Pages

- When first page frame is assigned to an anonymous region, a new **`anon_vma`** is created

- When a page frame already referenced by one process is inserted into a PTE of another process, the kernel simply inserts the anonymous memory region of the second process in the doubly linked circular list of the **`anon_vma`** data structure pointed to by the first process

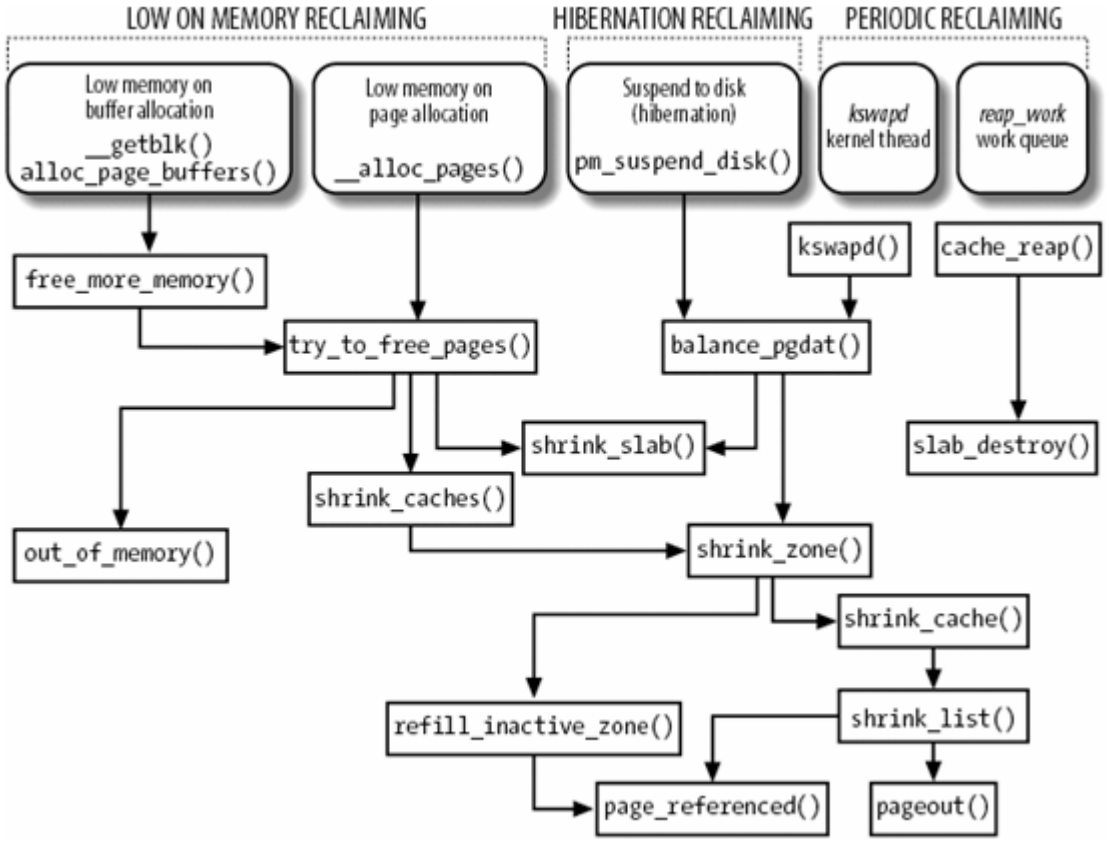
- To find all other PTEs for a page frame to be reclaimed is a linear traversal of the list

Reverse Mapping for Mapped Pages

- Similar to above except use **`address_space`** instead of **`anon_vma`**

- Earlier methods of using priority search tree did not seem to survive

Page Reclamation

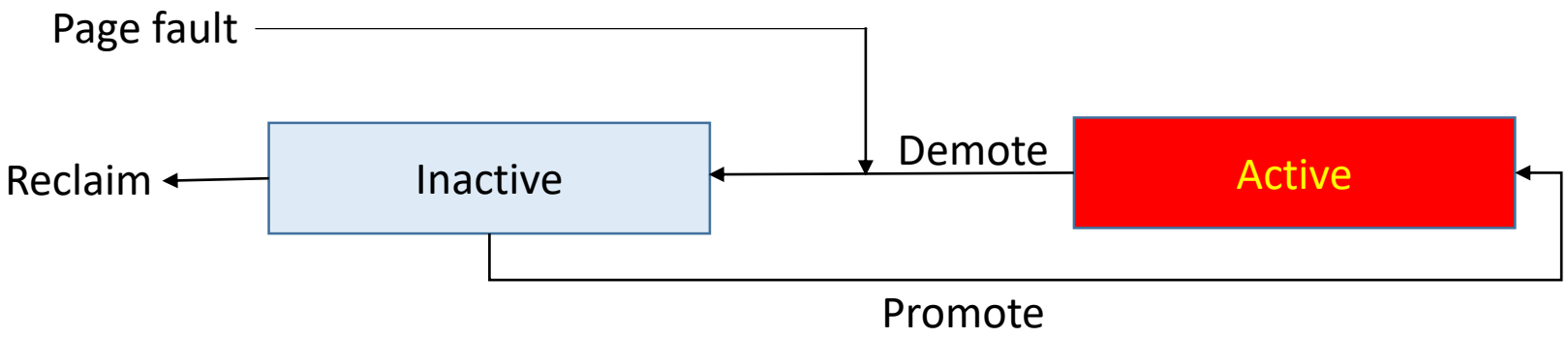


Source: Bovet, Daniel P. Understanding the Linux Kernel.

Page Reclamation

Page reclamation for mapped page frames

- Also known as demand paging
- For each memory zone, two LRU ("clock") lists – the active and inactive lists – of pages are maintained



Per zone, dual clock list

- Aim: to capture the working set of the process
- Freshly faulted pages start out at the head of the inactive list
- Page reclaimation scans pages from the tail of the inactive list
- Pages that are accessed multiple times on the inactive list are promoted to the active list
- Active pages are demoted to the inactive list when the active list grows too big
  - Once in the active list, one's position is not updated

Working set

- A workload is thrashing when its pages are frequently used but they are evicted from the inactive list every time before another access would have promoted them to the active list
- When the average access distance between thrashing pages is bigger than the size of memory there is nothing that can be done
- Otherwise, the scheme approximates the working set of the process

Overhead

- Too expensive to accurately track access frequency of pages
- A reasonable approximation can be made to measure thrashing on the inactive list

Observations

- When a page is accessed for the first time in the inactive list, it is added to the head of the inactive list, and pushes the current tail out of memory ("evicted")
- When it is accessed for the second time, it is promoted ("activated") to the active list, shrinking the inactive list by one slot

- The sum of evictions and activations in a given period of time indicates the minimum number of inactive pages accessed in this period
- Moving one inactive page N page slots towards the tail requires at least N inactive page accesses

Insights

- When a page is finally evicted from memory, the number of inactive pages accessed is at least the number of slots in the inactive list

- Measuring the sum of evictions and activations (E) at the time of a page's eviction, and comparing it to another reading ($R > E$) at the time the page faults back into memory tells the minimum number of accesses while the page was not cached
  - This is called the refault distance

Minimum Access Distance

- Minimum Access Distance = **`NR_inactive`** + $(R - E)$

- In other words, the refault distance can be seen as a deficit in inactive list space
  - If the inactive list had $(R - E)$ more page slots, the page would not have been evicted in between accesses, but activated instead

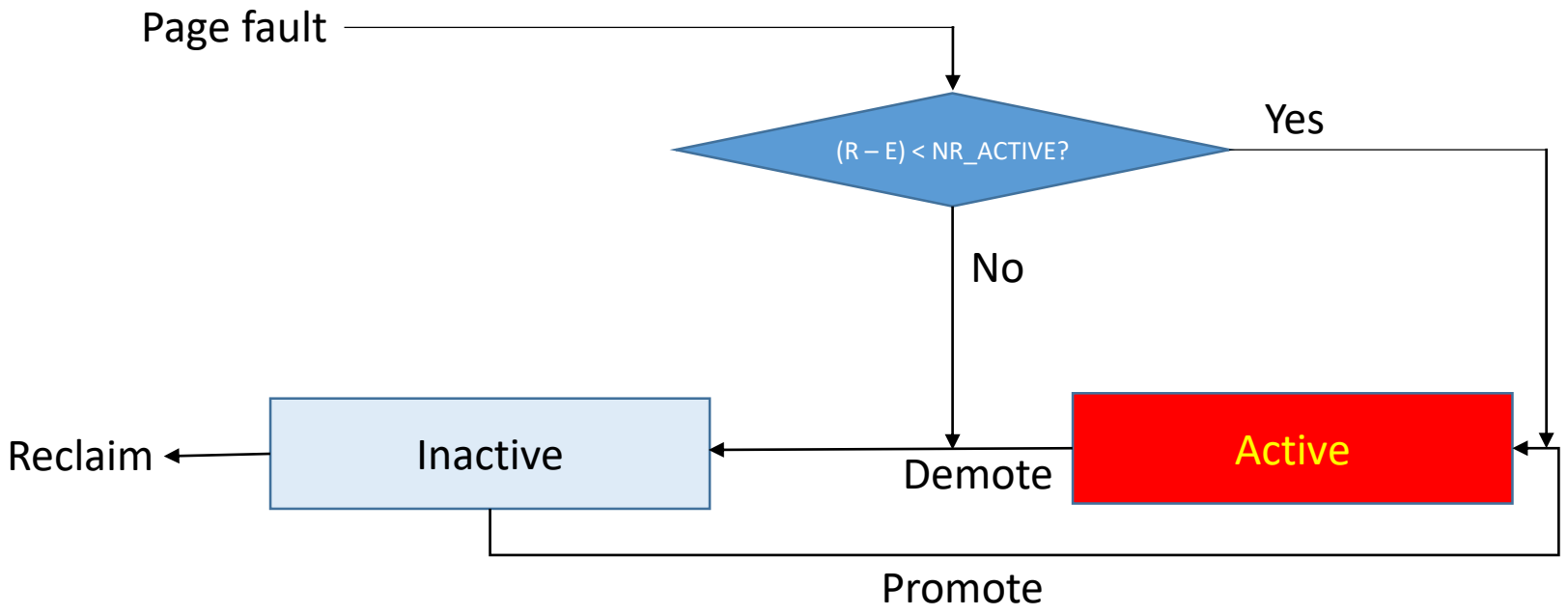
- Heuristic: place page in active list ("activate") if

$$(R - E) < \texttt{NR\_active}$$

Implementation

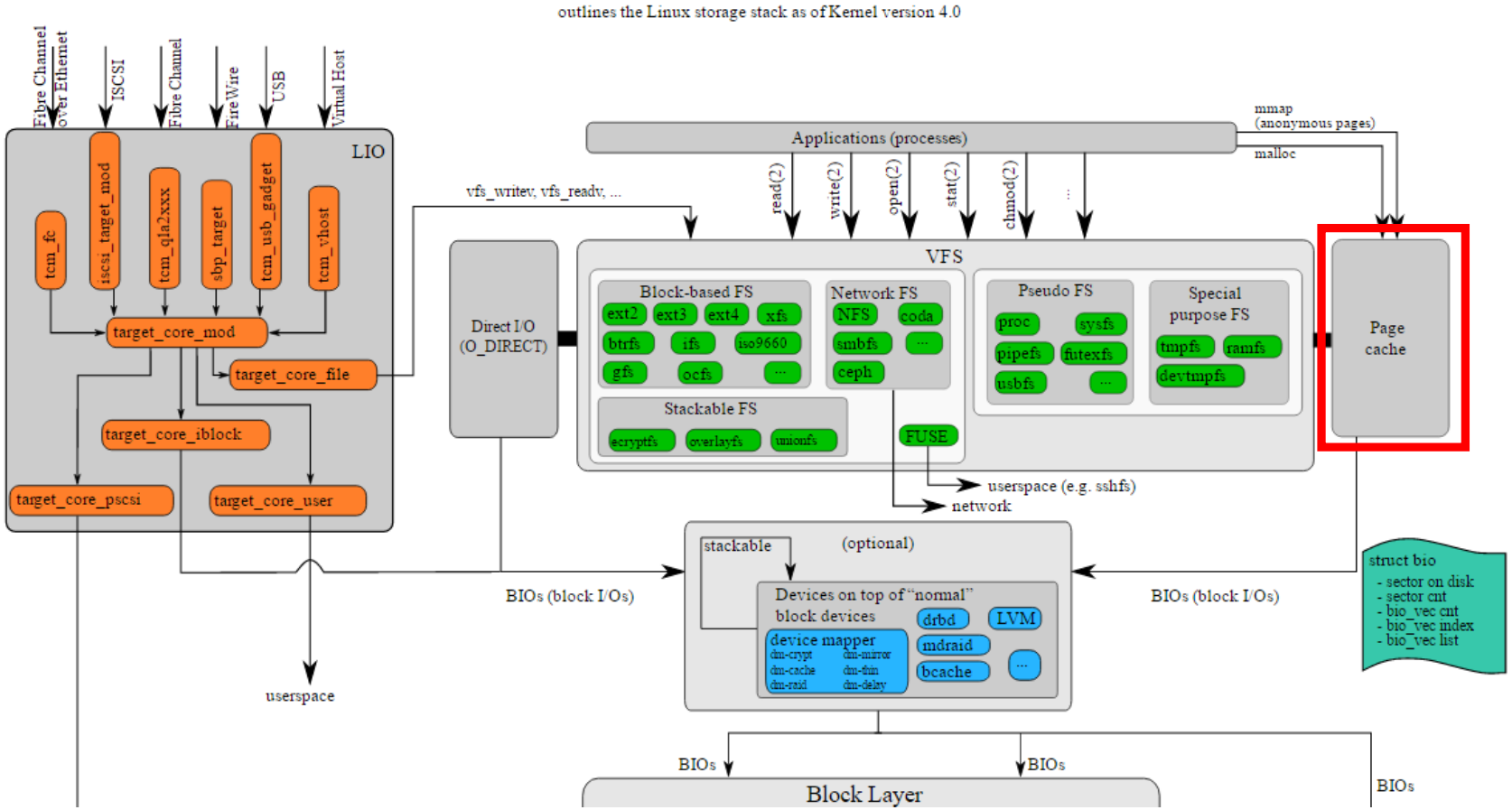
- Maintain a counter for inactive evictions and activations for each zone
- On eviction, a snapshot of this counter (along with some bits to identify the zone) is stored in the now empty page cache radix tree slot of the evicted page.
  - This is called a shadow entry
- On page cache misses for which there are shadow entries, an eligible refault distance will immediately activate the refaulting page.

Modified flow



The Page Cache

The Linux Storage Stack



The Page Cache

- Caches pages originating from reads and writes of regular filesystem files, block device files, and memory-mapped file
- It is the “bridge” between the memory and disks
- When the kernel tries to read from a file, it will check if it is in the page cache.
  - If so, it will return those pages.
  - If not, schedule an I/O operation that will read from (parts of) disk files into the page cache

History

- Before kernel 2.2, there exist a page cache and a buffer cache
  - Page cache for pages, buffer cache for VFS blocks
  - Blocks are unit of transfer for block devices
    - Block size is device dependent and may not be the same as the page size

- Since kernel 2.4, there is only the page cache
  - Has instead buffer page - a page of data additional descriptors called “buffer heads ,” for quick location of disk address of each individual block in the page

Write Caching

- Linux uses write-back caching
- On a file write request, data is written into the page cache. Those written to pages are marked dirty
- Periodically, dirty data is written back to disk (eviction)

`address_space` – the page frames of a file

```
struct address_space {
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root  page_tree;      /* radix tree of all pages */
        spinlock_t              tree_lock;      /* and lock protecting it */
        atomic_t                i_mmap_writable;/* count VM_SHARED mappings */
        struct rb_root          i_mmap;         /* tree of private and shared mappings */
        struct rw_semaphore     i_mmap_rwsem;   /* protect tree, count, list */
        /* Protected by tree_lock together with the radix tree */
        unsigned long           nrpages;        /* number of total pages */
        /* number of shadow or DAX exceptional entries */
        unsigned long           nrexceptional;
        pgoff_t                 writeback_index;/* writeback starts here */
        const struct address_space_operations *a_ops;   /* methods */
        unsigned long           flags;          /* error bits/gfp mask */
        spinlock_t              private_lock;   /* for use by the address_space */
        struct list_head        private_list;   /* ditto */
        void                    *private_data;  /* ditto */
} __attribute__((aligned(sizeof(long))));
```

Noteworthy fields

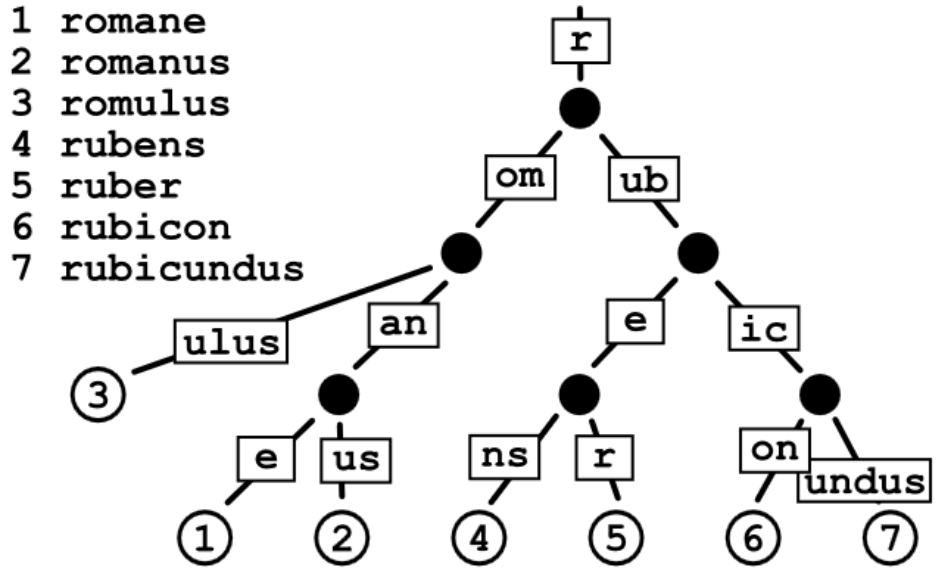
- **`i_mmap`**: all the memory mappings (private or shared) captured in a red-black tree

- **`page_tree`**: for fast query about the existence of a page in the page cache (coz files can be huge and only a small portion of pages are cached)
  - Organized as a radix tree

General Radix Tree Example

- Wikipedia: Radix trees are useful for constructing associative arrays with keys that can be expressed as strings

- Linux: used for looking up integers



Source: Wikipedia

Radix tree from addresses example

0x12: 0000 0000 0000 0001 0010

0_{10} 18_{10}

0x14: 0000 0000 0000 0001 0100

0_{10} 20_{10}

0xbc: 0000 0000 0000 1011 1100

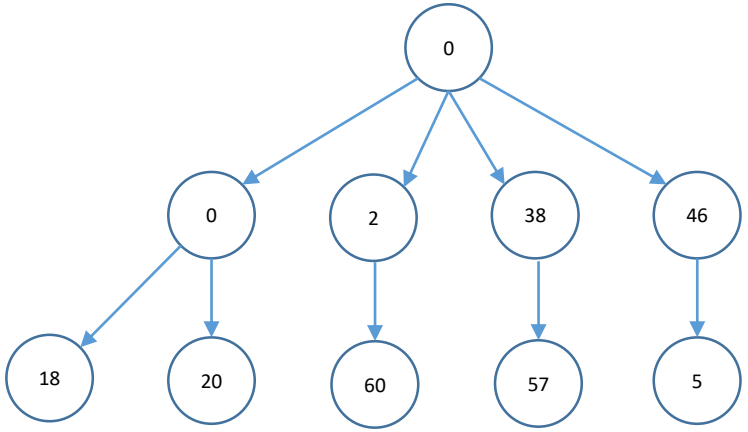
2_{10} 60_{10}

0x9b9: 0000 0000 1001 1011 1001

38_{10} 57_{10}

0xb85: 0000 0000 1011 1000 0101

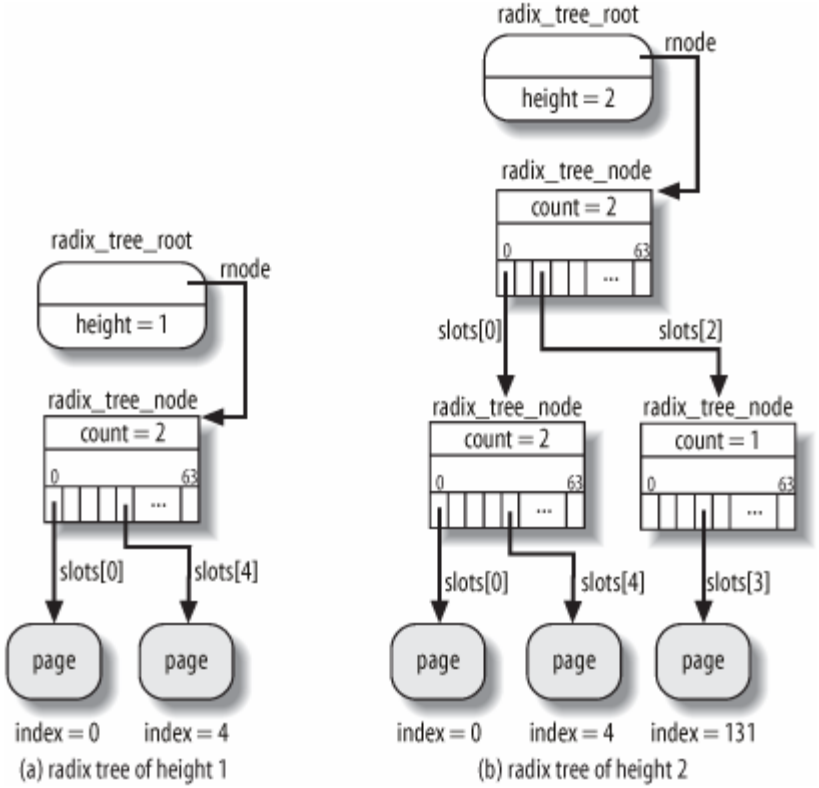
46_{10} 5_{10}



Radix tree in Linux

- Page index = position of a page inside the file

Radix tree height	Highest index	Maximum file size
0	none	0 bytes
1	2^6 -1 = 63	256 kilobytes
2	2^{12} -1 = 4 095	16 megabytes
3	2^{18} -1 = 262 143	1 gigabyte
4	2^{24} -1 = 16 777 215	64 gigabytes
5	2^{30} -1 = 1 073 741 823	4 terabytes
6	2^{32} -1 = 4 294 967 295	16 terabytes

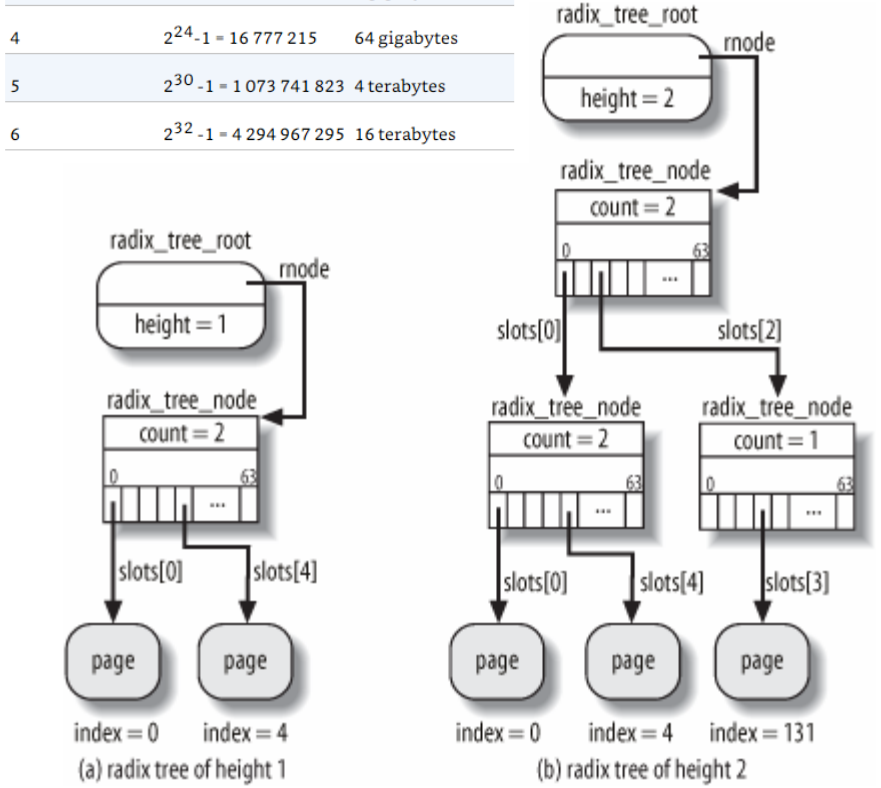


Source: Bovet, Daniel P. Understanding the Linux Kernel.

Radix tree in Linux

- Treat the bits in the page index as a “bit string”
- Take 6-bits at a time
  - Level 1 – bits 0-5
  - Level 2 – bits 6-11
  - Level 3 – bits 12-17
  - Level 4 – bits 18-23
  - Level 5 – bits 24-31
  - Level 6 – bits 32-37
- Either null (not found), or a next level internal node, or a page descriptor pointing to the page (found)

Radix tree height	Highest index	Maximum file size
0	none	0 bytes
1	2^6 -1 = 63	256 kilobytes
2	2^{12} -1 = 4 095	16 megabytes
3	2^{18} -1 = 262 143	1 gigabyte
4	2^{24} -1 = 16 777 215	64 gigabytes
5	2^{30} -1 = 1 073 741 823	4 terabytes
6	2^{32} -1 = 4 294 967 295	16 terabytes



Source: Bovet, Daniel P. Understanding the Linux Kernel.

Page Cache Eviction

- When space is low in the page cache, eviction is triggered
- First try to evict clean pages
- If not enough, force writeback of some dirty pages using LRU

Per-backing-device based writeback

- Prior to kernel 2.6.32, flushing to backing device was done by pdflush kernel thread

- Now, each backing device has dedicated kernel thread for flushing
  - “flush-MAJOR”
  - Threads are created when there's flushing work that needs to be done and will disappear after a while if there's nothing to do

Backing device

- Block device – transfer units of blocks
  - As opposed to “char devices” with transfer unit of a character
  - Different block devices have different block sizes
  - Examples: hard disks, tapes, etc.

- A backing device is a block device whose content may be cached in memory
  - Provides an abstract interface to get to the device driver to do common operations like read or write
  - Info for all available backing devices held in a list, **`bdi_list`**
  - Added writeback functionality to support page cache eviction

Swapping

The need to swap

- Pages that belong to an anonymous memory region of a process (User Mode stack or heap)
- Dirty pages that belong to a private memory mapping of a process
- Pages that belong to an IPC shared memory region

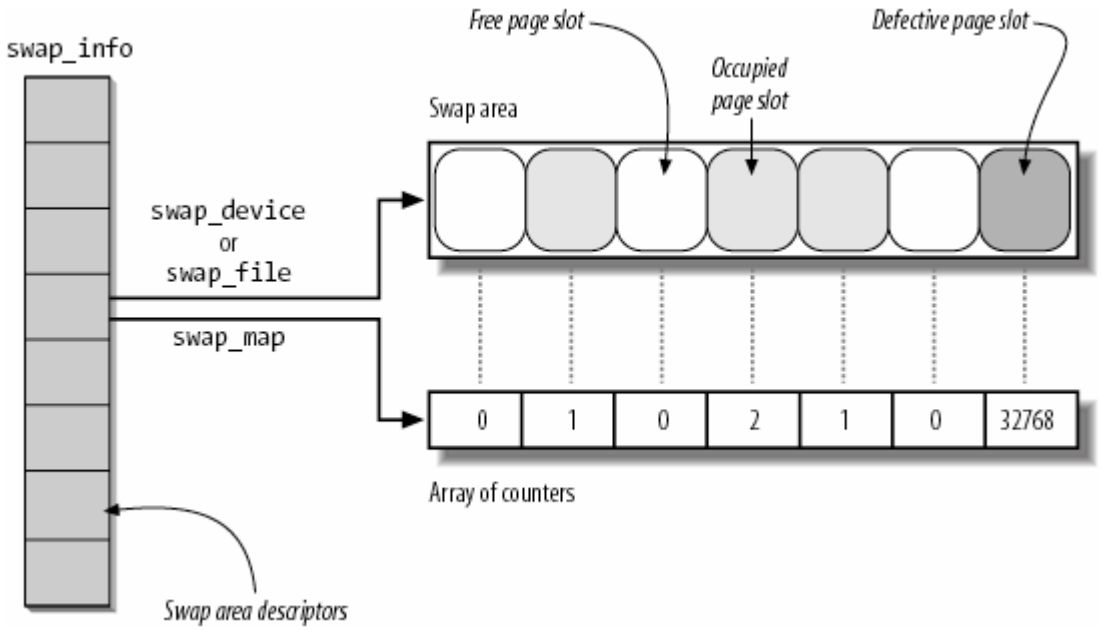
Work of the Swapping Subsystem

- Set up “swap areas” on disk to store pages that do not have a disk image.
- Manage the space on swap areas allocating and freeing “page slots” as the need occurs.
- Provide functions both to “swap out” pages from RAM into a swap area and to “swap in” pages from a swap area into RAM.
- Make use of “swapped-out page identifiers” in the Page Table entries of pages that are currently swapped out to keep track of the positions of data in the swap areas.

Swap Area

- May be a disk partition or a file included in a larger partition
- Up to **`MAX_SWAPFILES`** (usually 32) may be used
- Each swap area consists of a sequence of page slots
  - Each slot holds the content of a page, i.e., 4096 bytes
  - Page slots are organized into swap extents – a contiguous group that is also contiguous physically on disk
    - Normally, in a disk partition, one swap area is one swap extent while if in a file, a swap area consists of several swap extents
  - First page slot of a swap area is used to contain meta information

Data structure of a swap area



Source: Bovet, Daniel P. Understanding the Linux Kernel.

Swap-out Page Identifier

- A page that is swapped out is identified by:
  - The swap area it is in
    - 7 bits at the moment
  - The page slot index
  - Usually stores as an “unsigned long”
    - Actual size is architecture dependent

Race conditions during swapping

- Multiple swap-ins: Two processes may concurrently try to swap in the same shared anonymous page.
- Concurrent swap-ins and swap-outs: A process may swap-in a page that is being swapped out by the PFRA.
- Solution: swap cache

The Swap Cache

- A collection of page frames

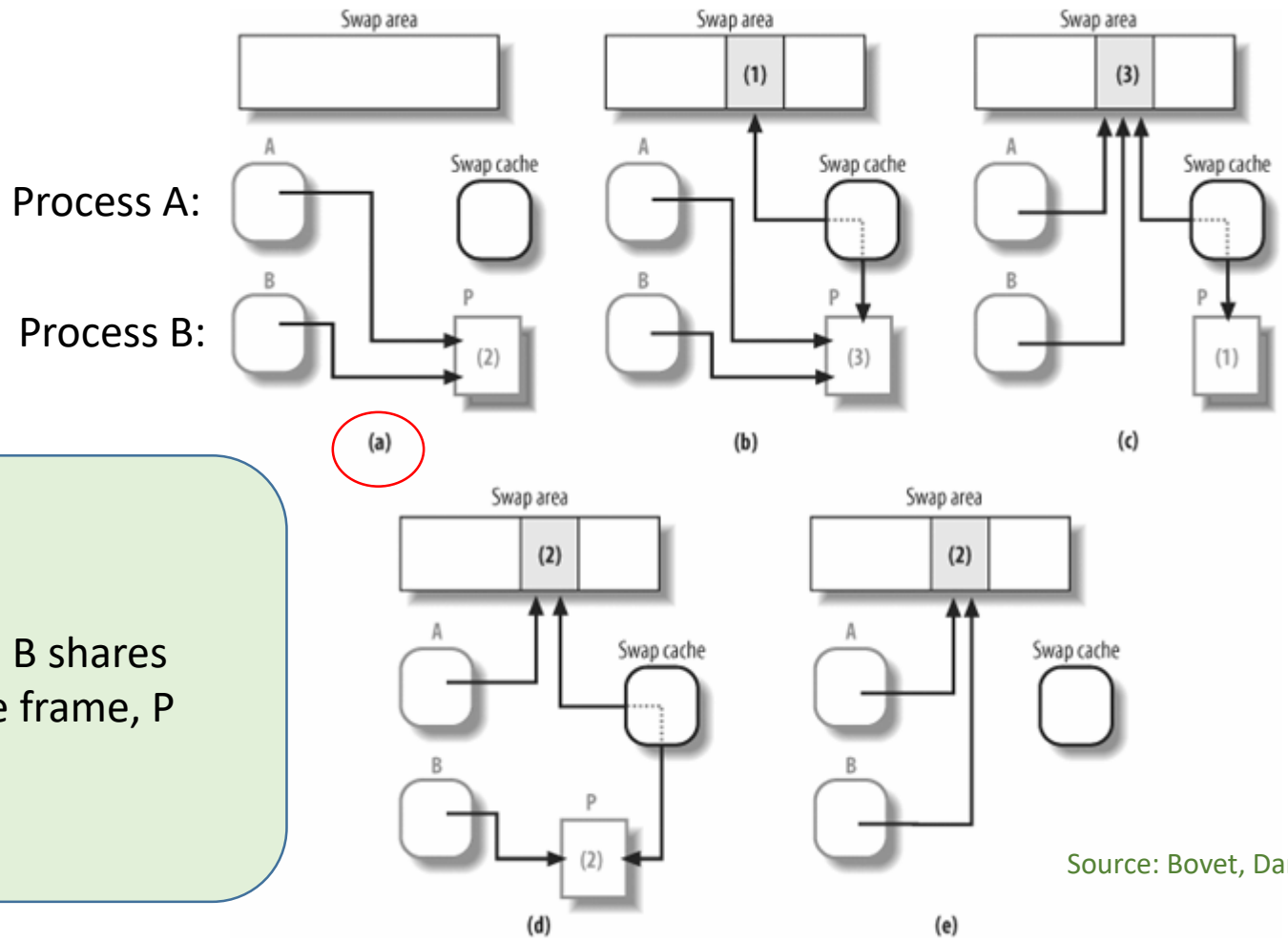
- Key rule: nobody can start a swap-in or swap-out without checking whether the swap cache already includes the affected page
  - Pages in swap caches are in page frames
  - If in swap cache, get a lock for the corresponding page frame

How the swap cache resolves races

- Process A: wants to swap in a page
  - Kernel checks if page is in swap cache; suppose it is not
  - Kernel allocates a new page frame for the incoming page and mark it locked

- Process B: wants to swap in same page
  - Kernel checks if page is in swap cache – it is, and it is locked
  - Process B goes to sleep waiting for page to be ready

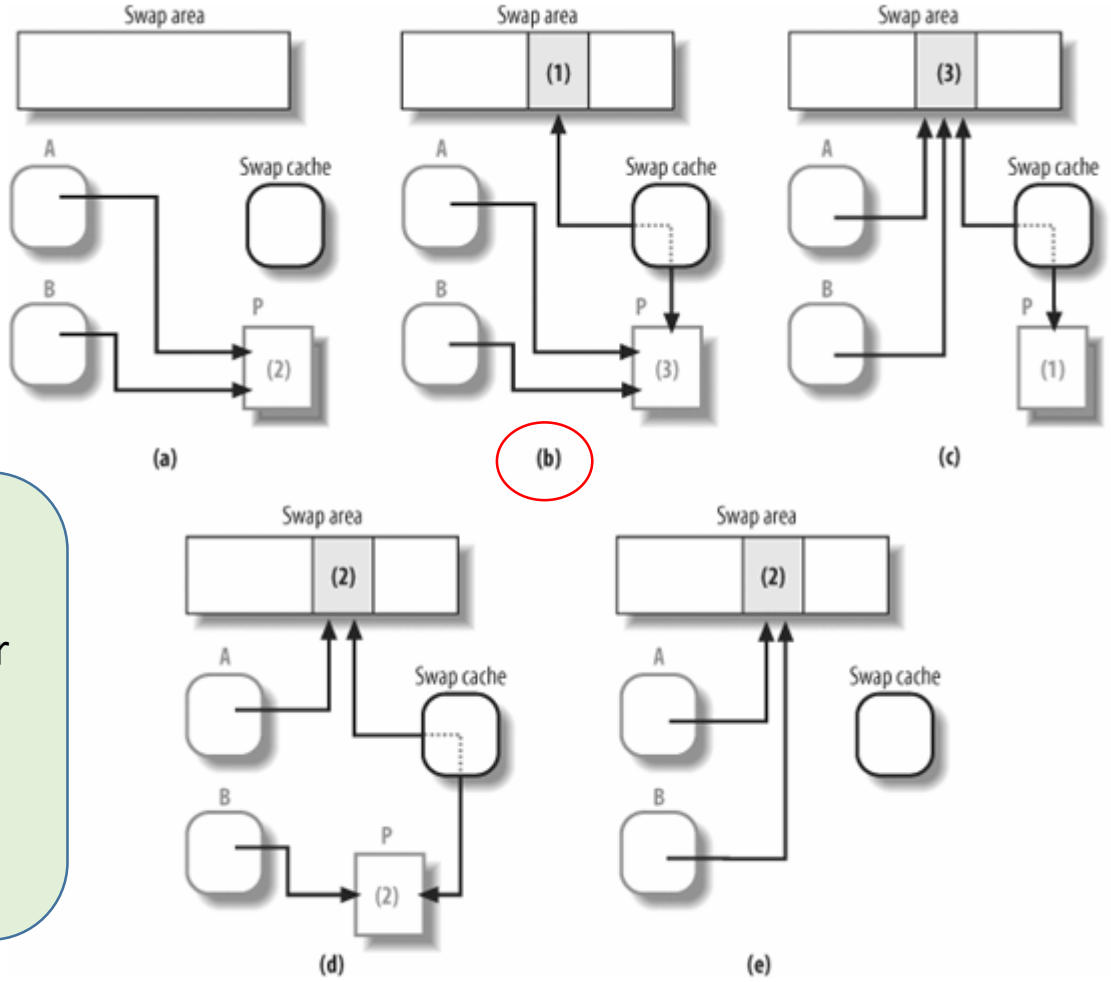
Page reclamation and the Swap Cache



Process A and B shares the same page frame, P

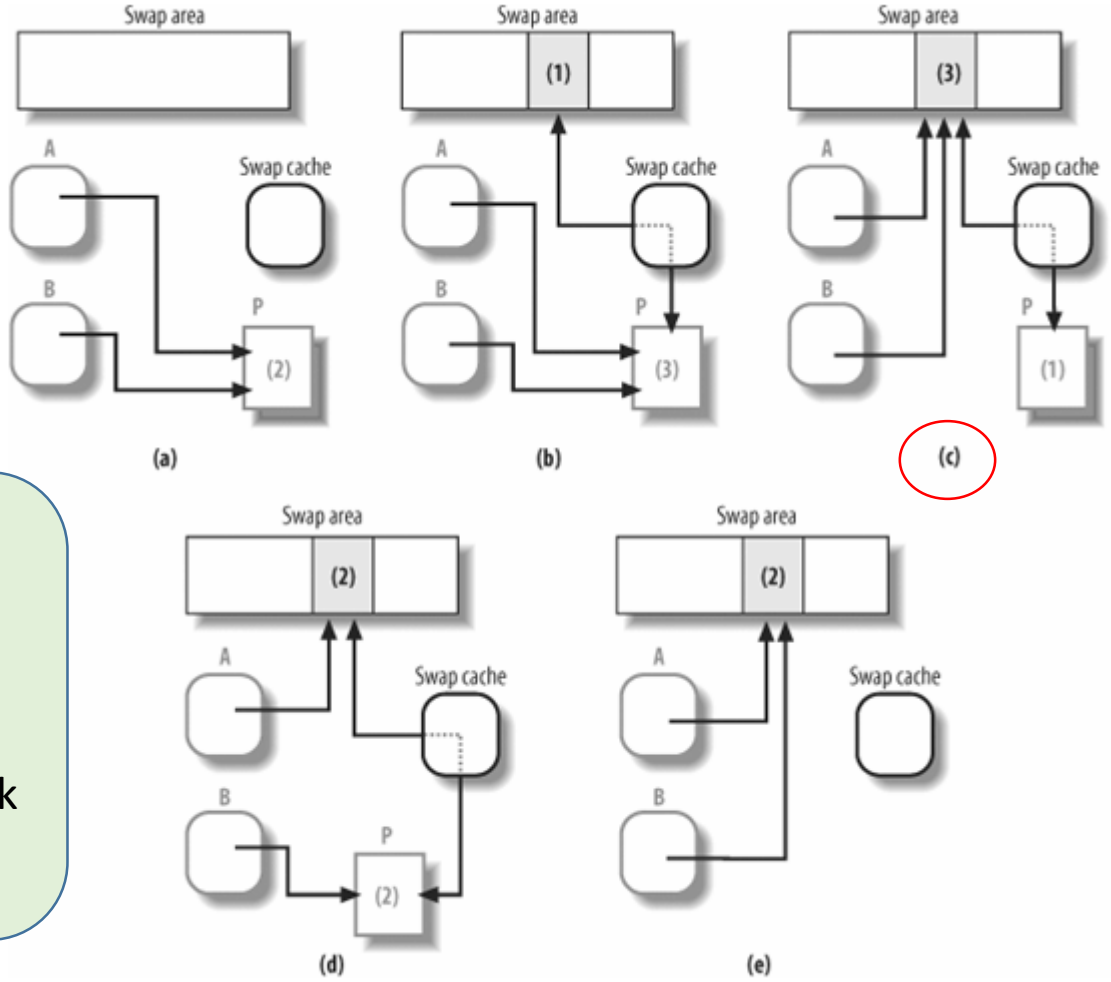
Source: Bovet, Daniel P. Understanding the Linux Kernel.

Page reclamation and the Swap Cache



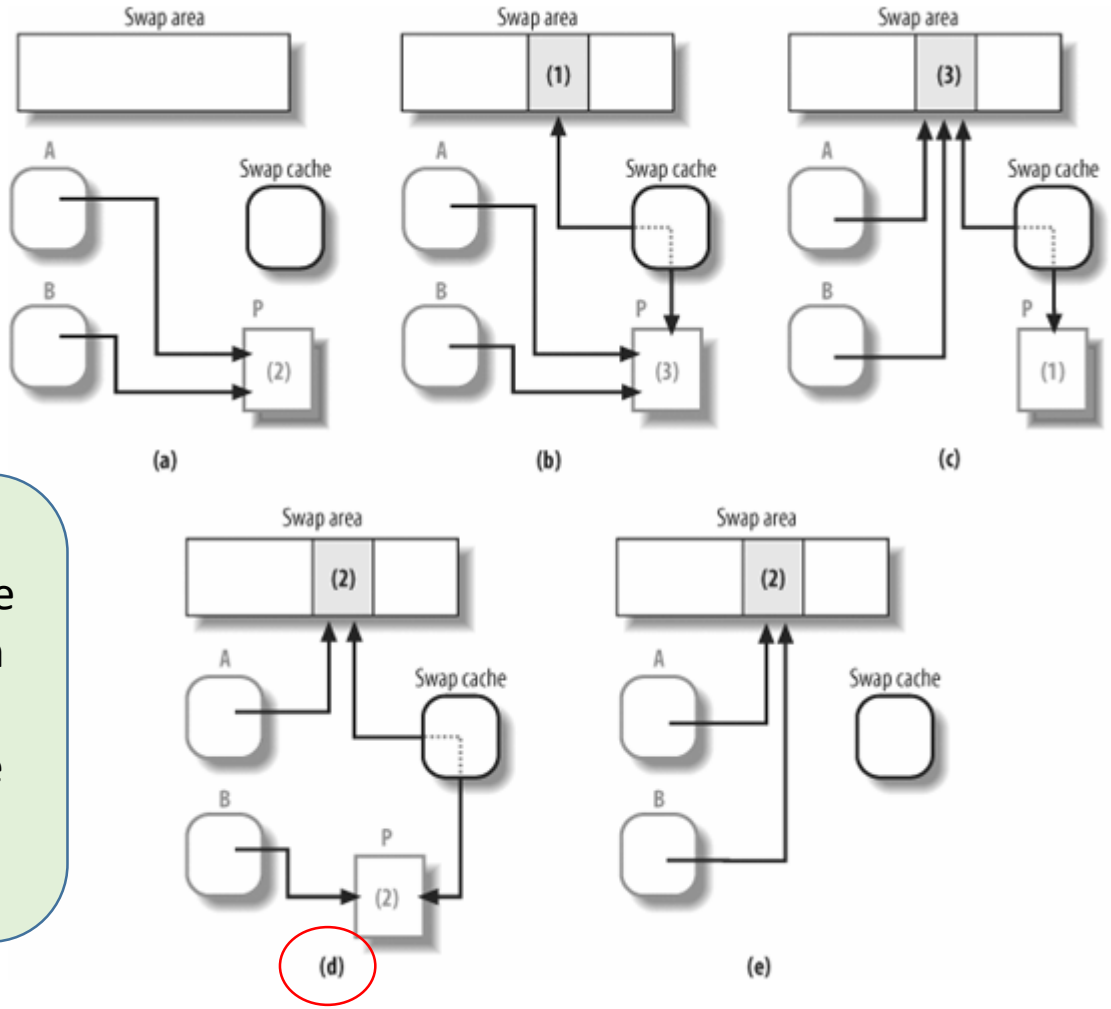
PFRA selects page frame P for reclamation; inserts page frame into swap cache

Page reclamation and the Swap Cache



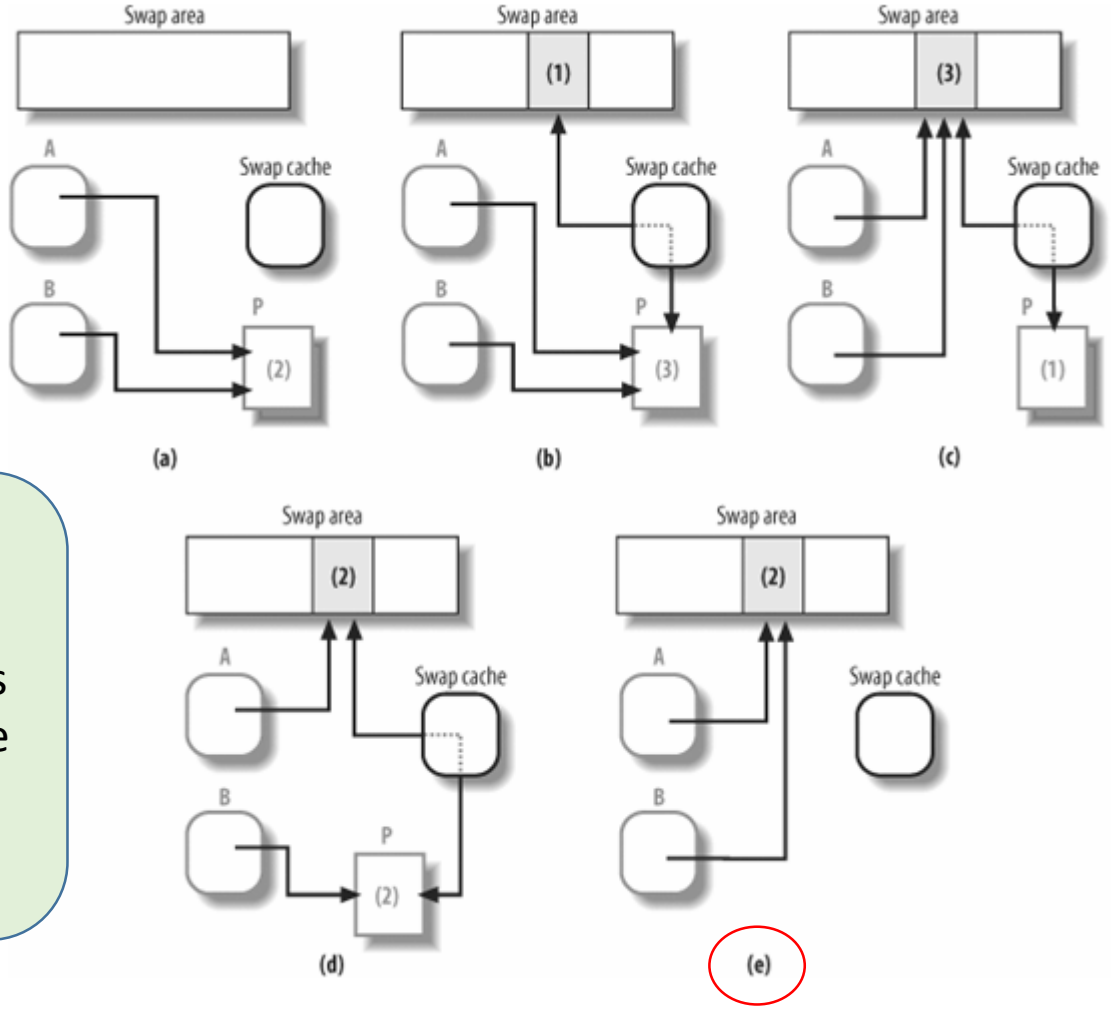
PFRA marks both Process A and B's PTE to point to swap area page slot. PFRA starts writing content of page to disk

Page reclamation and the Swap Cache



In the middle of it all, suppose Process B wants to read from the page. Page fault handler will find the page in the page cache and direct B to it.

Page reclamation and the Swap Cache



However, if write completes without interference, Process A and B will be pointed to the page slot

Allocating and Freeing Page Slots in the Swap Area

- Swapping (because of high disk access overhead) is batched
- Allocation will be for a batch of pages
  - The pages themselves may or may not be contiguous in virtual memory
  - But they will sit in contiguous space
  - A “first used” and “first free” pointer is maintained in the swap area info
  - Scans from these to find free slots

Swapping In Pages

1. Check swap cache, if page is there, skip to Step 3
2. Read in pages from disks
   a. For each page, check page cache by looking up radix tree, check swap cache before trying to read from disk
3. Check page cache by looking up radix tree, check swap cache again – to make sure we got all pages in correctly
   a. Tricky race conditions and various technical issues can trip up 2
4. Free swap area page slots
5. Fix up page table entries of process

Swapping out pages

- Insert pages into swap cache
- Get allocation of swap area page slots
- Update page table entries
- Write pages into swap area

Regular maintenance

- **`kswapd`** – Kernel Swap Daemon
  - Periodically frees a number of pages each time it runs
    - Don’t free too many! That will create lots of I/O

- **`cache_reap`**
  - Called periodically to clear pages in the slab

zswap

- A compressed write-back cache for swapped pages
  - Memory is getting big these days
  - I/O to disk is still slow
  - Trade I/O with computation and memory

- Instead of moving memory pages to a swap device when they are to be swapped out, zswap compresses and then stores them into a memory pool dynamically allocated in the system RAM

Kernel Samepage Merging

Memory De-duplication

Motivation

- Page frames are precious resource
- Virtual machines may have lots of common pages
- There is no point in duplicating them when they can be shared
  - Only doable for read-only
  - For potentially writable page that is for the moment the same, rely on COW

How Linux does it

- Merged pages may be paged out. If so, have to be rediscovered
- Maintain two red-black trees: stable and unstable
  - Stable tree holds pointers to write-protected merged pages
  - Unstable tree holds pointers to pages found to be “unchanged over a period of time”
    - Not write-protected, can be changed any time
    - The unstable tree is flushed at the end of every scan cycle
- One more data structure to remember the old hash value associated with a particular virtual address

Constraints

- Works only on anonymous pages, never on page cache

- Only operates on those areas of address space which an application has advised to be likely candidates for merging, by using system call: **`madvise(addr, length, MADV_MERGEABLE)`**

Scanning

- KSM daemon **`ksmd`** periodically scans those areas of user memory which have been registered with it, looking for pages of identical content which can be replaced by a single write-protected page

- Use jhash2() to obtain a hash of a page
  - Two pages having the same hash key are deemed the same
  - **`jhash()`** hashes an array of bytes, **`jhash2()`** hashes an array of unsigned 32 bit words

Scanning steps

1. Compute hash of current page (let's call it H) using **`jhash2()`**

2. Check the stable tree. If found, add an additional sharing reference.

3. Check the unstable tree.
   a) If H is not found in the unstable tree
      i. Look for old hash value associated with the virtual address of the page
      ii. If old hash is different, update hash value stored and continue (do not insert!)
      iii. Else insert and continue
   b) If H is found in the unstable tree:
      i. Change to "write-protected"
      ii. Move to stable tree

End