Linux Kernel : Memory Management

Linux Kernel : Memory Management

Virtual Memory

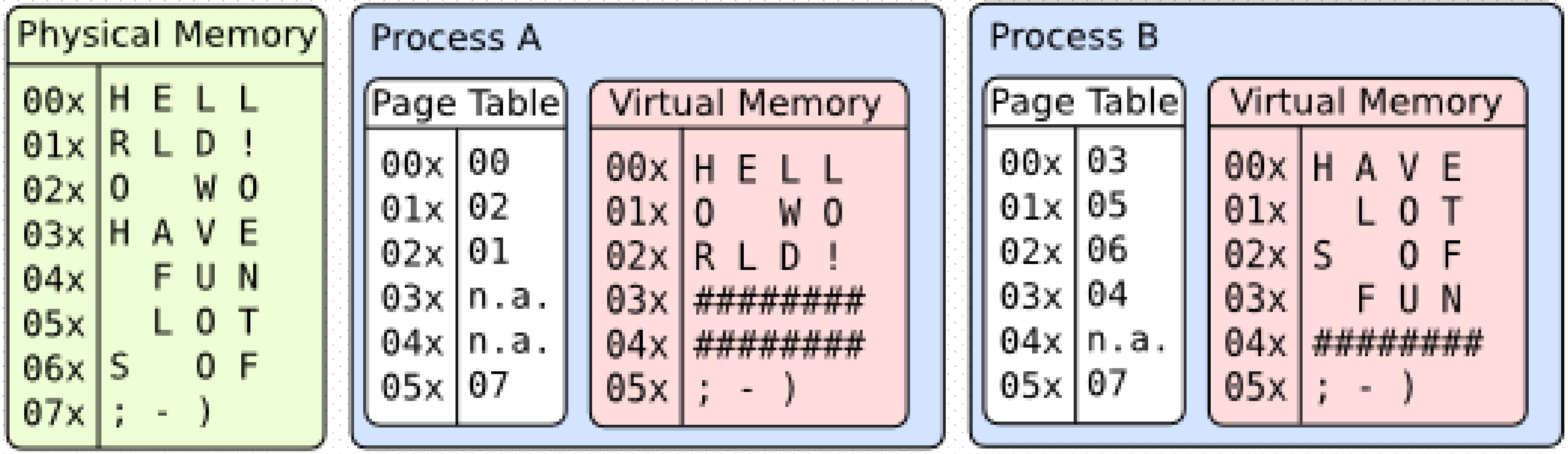
- ➔ **Introduction**
- ➔ **The How**
- ➔ **The Why**

Why VM?

- **Memory is virtualized**
  - Programmer does not worry about phy RAM, availability, etc
- **Information Isolation**
  - Each process run's in it's own private VAS
- **Fault Isolation**
  - One process crashing will not cause others to be affected
  - Multithreaded (firefox) vs multiprocess (chrome)

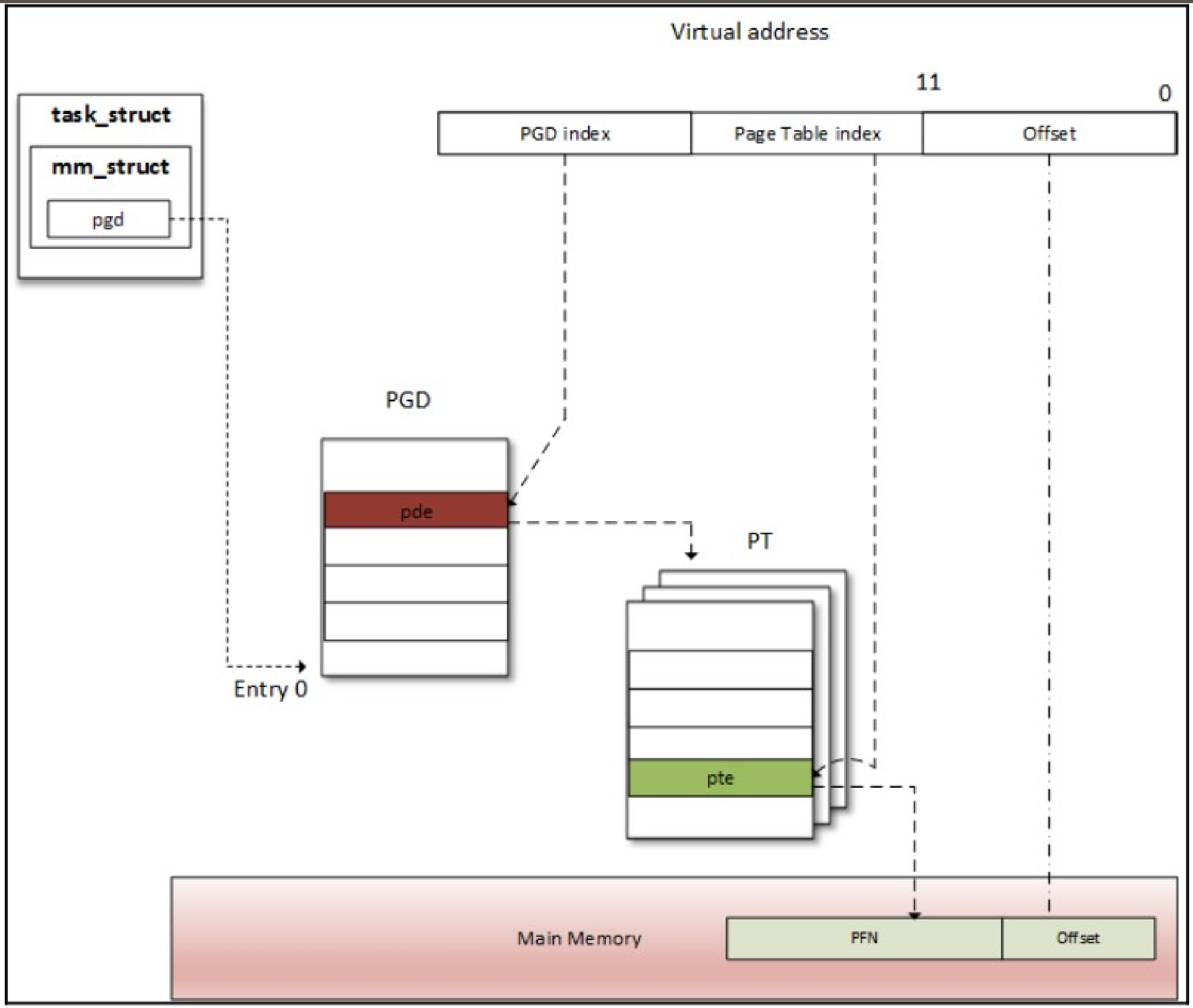
Linux Kernel : Memory Management

Paging Tables – a *very* simplistic and conceptual view



Source

Page Table Layout on the x86_64



Source: Linux Device Driver Development, John Madieu, Packt, 2017

Page Table Layout on the x86_64

Source: ULK3

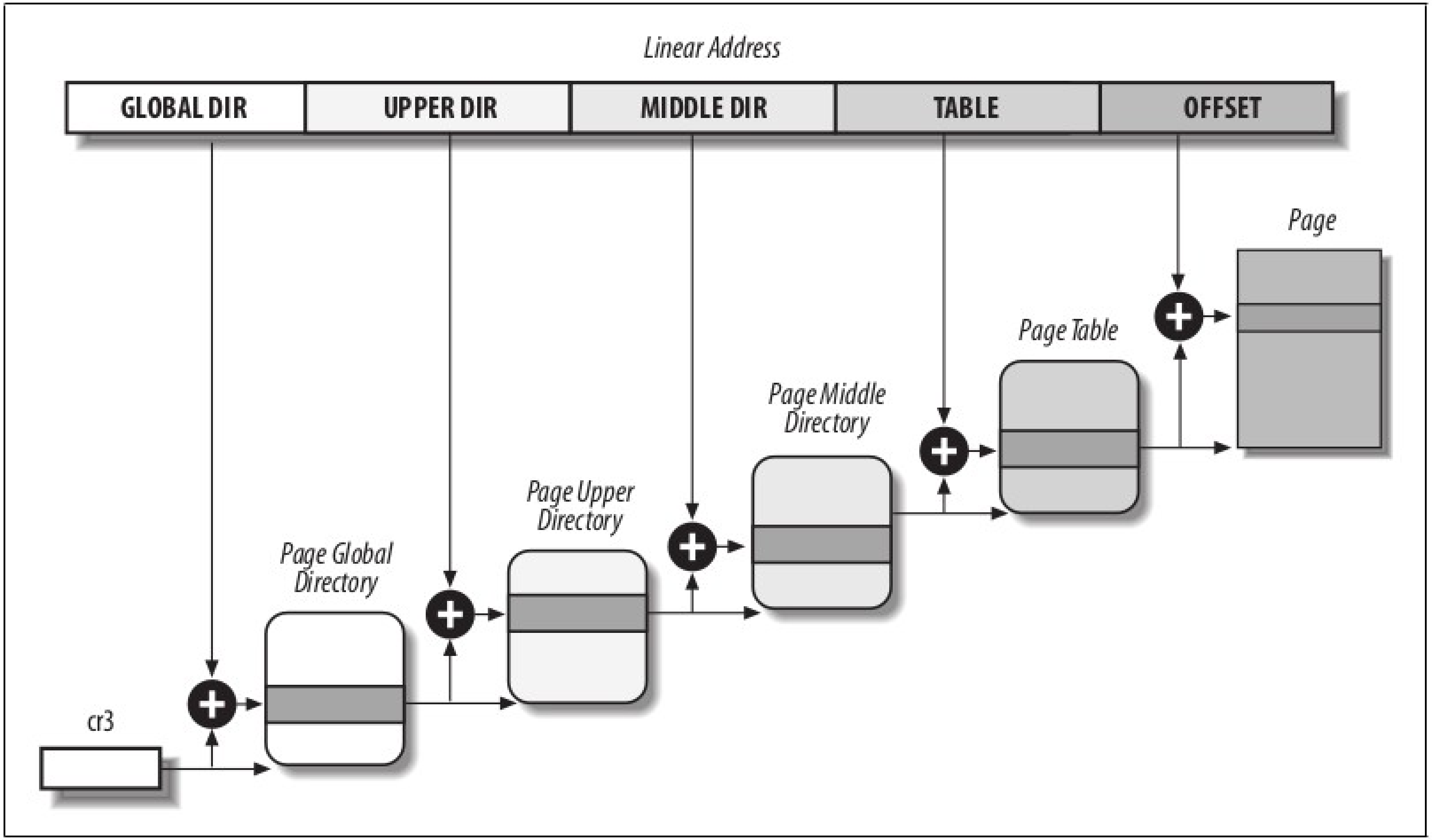


Figure 2-12. The Linux paging model

Page Table Layout on the x86_64

4-level Paging on x86_64

Article: Page Table Layout

```
6   65               55555          433            32          22          11                0
3210987654321098765432109876543210987654321098765432109876543210
[      Unused:16      ][   PGD:9   ][   PUD:9   ][   PMD:9   ][   PTE:9    ][   offset:12  ]
```

[...]
Eg. (virtual) address:
00000000 00000000 00000000 00000000 00000000 10110001 01110000 00010000

```
      00          00         00          00          00          b1          70          10
```

0xb17010

And so we can now visualise what a given example address (actually represents:

```
    6          5          4          3          2          1
4321098765432109876543210987654321098765432109876543210987654321
0000000000000000000000000000000000000000101100010111000000010000
[    RESERVED   ][  PGD   ][   PUD  ] [   PMD  ][   PTE  ][  OFFSET   ]
                           |          |         |         |-PAGE_SHIFT
                           |          |         |------------PMD_SHIFT
                           |          |----------------------PUD_SHIFT
                           |--------------------------------PGDIR_SHIFT
```

But how does this relate to a 'sparse' layout of pages,
what do these offsets actually reference?

Page Table Layout on the x86_64

4-level Paging on x86_64

Article: Page Table Layout

The best way of showing how this fits together is diagrammatically, so taking the example address from above:

```
0000000000000000000000000000000000000000000000101100010111000000010000
[     RESERVED      ][    PGD    ][    PUD    ][   PMD    ][    PTE     ][    OFFSET    ]
```

PGD offset = 000000000 = 0

PUD offset = 000000000 = 0

PMD offset = 000000101 = 5

PTE offset = 100010111 = 279

phy offset = 000000010000 = 16

Page Table Layout on the x86_64

4-level Paging on x86_64

[Article: Page Table Layout]

```
       PGD
     -------
   0 |    -----\        PUD
   . |-----|   |      -------
   . /     /   \--->0 |    -----\        PMD
   . \     \        . |-----|   |      -------
   . /     /        . /     /   \--->0 /     /
   . \     \        . \     \        . \     \
   . /     /        . /     /        . /     /
   . \     \        . \     \        . |-----|
   . /     /        . /     /        5 |    -----\        PTE
 512 -------        . \     \        . |-----|   |      -------
                    . /     /        . /     /   \--->0 /     /
                  512 -------        . \     \        . \     \
                                     . /     /        . /     /
                                   512 -------        . |-----|
                                                    279 |    -----\   Phys Page
                                                      . |-----|   |      ---
                                                      . /     /   \--->0 / /
                                                      . \     \        . \ \
                                                      . /     /        . / /
                                                    512 -------        . |-|
                                                                      16 |o|
                                                                       . |-|
                                                                       . |h|
                                                                       . |-|
                                                                       . |a|
                                                                       . |-|
                                                                       . |i|
                                                                       . |-|
                                                                       . |!|
                                                                       . |-|
                                                                       . / /
                                                                       . \ \
                                                                       . / /
                                                                    4096 ---
```

0000000000000000000000000000000000000000000000000000101100010111000000010000

[RESERVED][PGD][PUD][PMD][PTE][OFFSET]

Address Transalation : ARM64

Address Translation on the ARMv8

Source (pg 12-3)

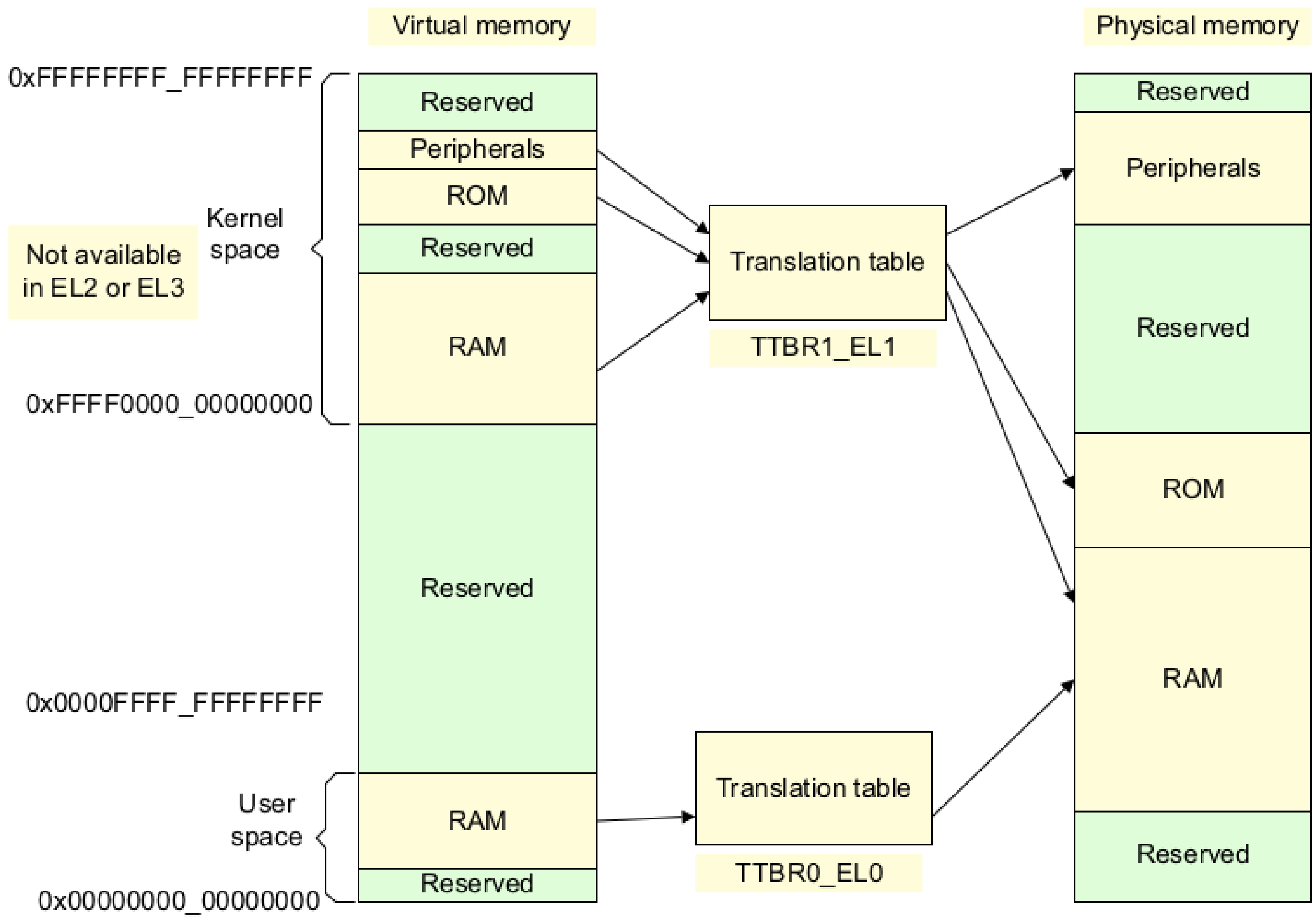
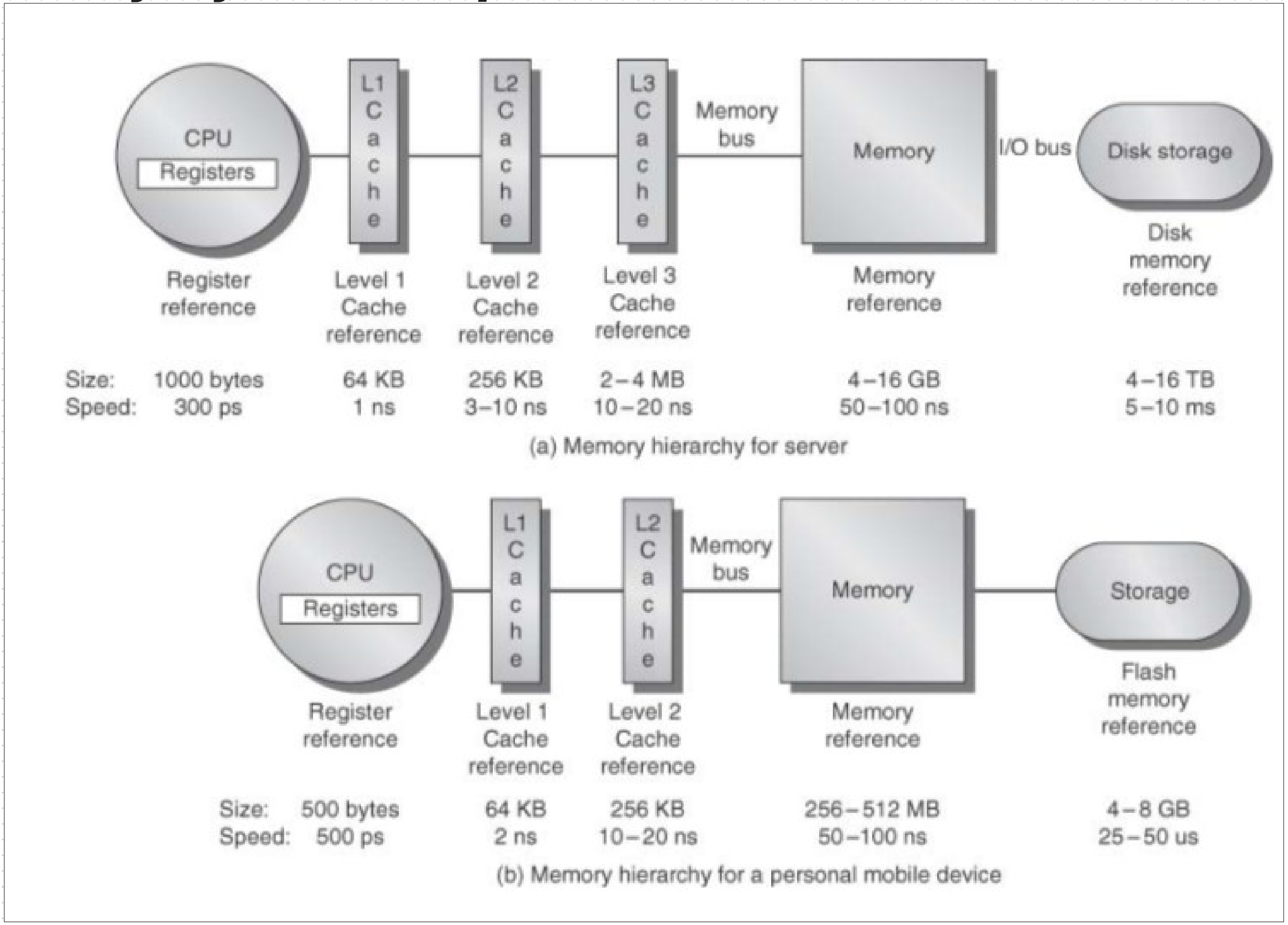


Figure 12-3 Address translation using translation tables

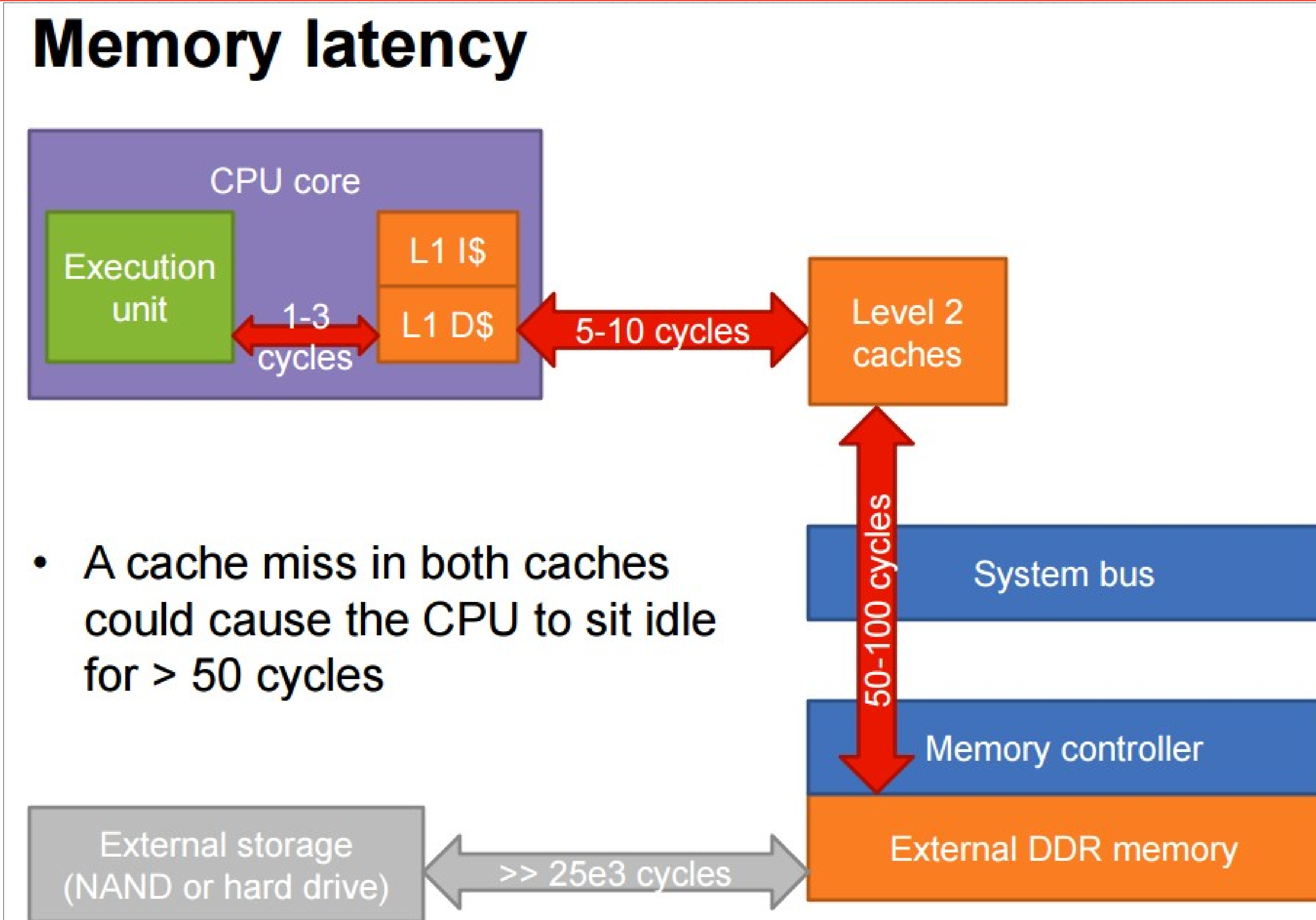
Linux Kernel : Memory Management

Memory Pyramid ; *Speed* and Size



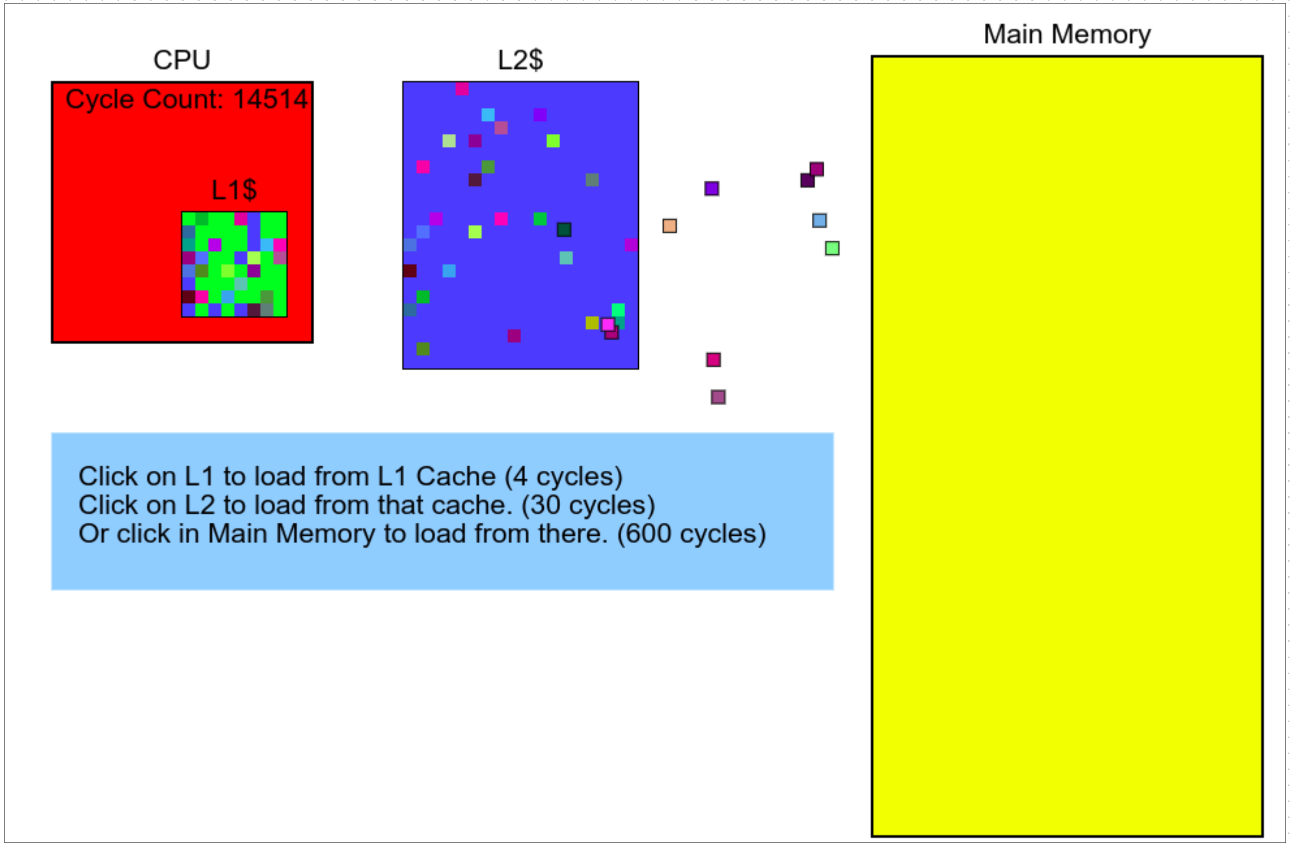
Computer Architecture, A Quantitative Approach, 5th Ed by Hennessy & Patterson.

CPU Caches


Memory latency
CPU core
Execution unit
L1 I$
L1 D$
1-3 cycles
5-10 cycles
Level 2 caches
50-100 cycles
System bus
Memory controller
External DDR memory
• A cache miss in both caches could cause the CPU to sit idle for > 50 cycles
External storage (NAND or hard drive)
>> 25e3 cycles

CPU Caches

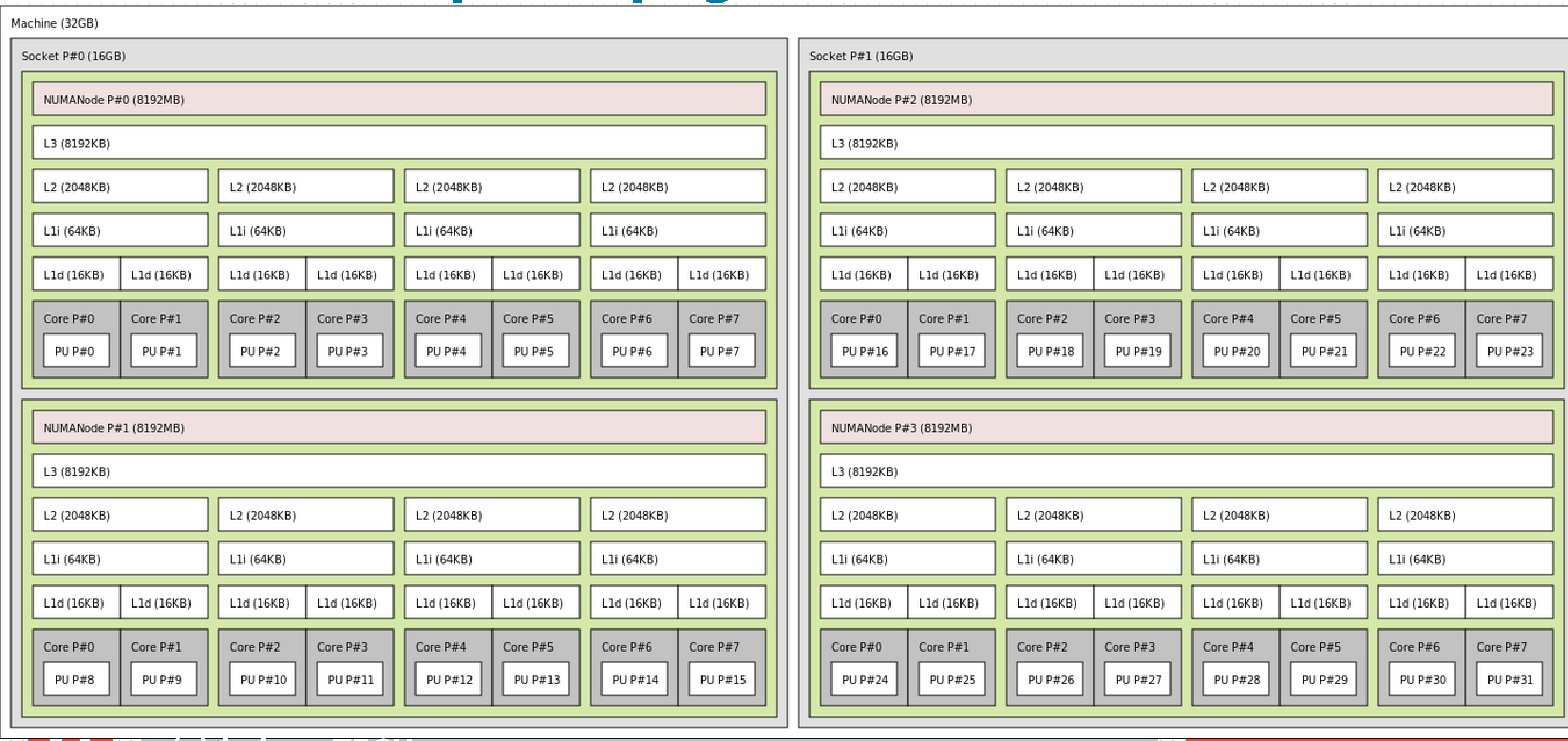
MUST SEE and TRY (it's interactive) !!!



http://www.overbyte.com.au/misc/Lesson3/CacheFun.html

Linux Kernel : Memory Management

An example (pic) from the **Wikipedia page on CPU Cache***:*



Linux Kernel : Memory Management

Software developers would do well to be aware how programming in a cache-aware manner can greatly optimize code:

- keep all important structure members ('hotspots') together and at the top
- ensure the structure memory start is CPU cacheline-aligned
- don't let a member "fall off" a cacheline (use padding if required; compiler can help)
- an LLC (Last Level Cache) Miss is expensive!
- use profilers (perf) to measure

[***Good Article:***

[How L1 and L2 CPU caches work, and why they're an essential part of modern chips, Joel Hruska, Feb 2016]

Also, careful of cache coherence issues; see these Wikipedia animations.

<< ***Lookup 1LinuxMM.pdf for details on:***

Hardware Paging: n-Level Paging, Linux's 4 (or now 5)-level arch-independent paging model, IA-32, ARM-32, x86_64 hardware paging, etc

>>

Linux Kernel : Memory Management

Virtual Address Space Splitting

- **Recall that Linux is a *monolithic OS***
- **The reality is that every process has its own private VAS, and all of them share a "common mapping"for the upper region – the kernel segment!**
- **The *typical* "VM split"**

  On a 32-bit x86 (IA-32) Linux system
  3 GB : 1 GB :: user-space : kernel-space

  On a 32-bit ARM Linux system
  2 GB : 2 GB :: user-space : kernel-space ***[sometimes***

Linux Kernel : Memory Management

Common powers of 2 : kilobyte onwards

Name (Symbol)	Value		Name (Symbol)	Value
kilobyte (kB)	10^3	2^{10}	kibibyte (KiB)	2^{10}
megabyte (MB)	10^6	2^{20}	mebibyte (MiB)	
gigabyte (GB)	10^9	2^{30}	gibibyte (GiB)	2^{30}
terabyte (TB)	10^{12}	2^{40}	tebibyte (TiB)	2^{40}
petabyte (PB)	10^{15}	2^{50}	pebibyte (PiB)	2^{50}
exabyte (EB)	10^{18}	2^{60}	exbibyte (EiB)	2^{60}
zettabyte (ZB)	10^{21}	2^{70}	zebibyte (ZiB)	2^{70}
yottabyte (YB)	10^{24}	2^{80}	yobibyte (YiB)	2^{80}

PAGE_OFFSET

Splitting point is called PAGE_OFFSET

- location from which physical RAM is direct-mapped (1:1) into kernel VAS

On an IA-32:

```
$ zcat /proc/config.gz |grep -i VMSPLIT
CONFIG_VMSPLIT_3G=y
# CONFIG_VMSPLIT_2G is not set
# CONFIG_VMSPLIT_1G is not set
$
```

Tip: Numbers to recognize : PAGE_OFFSET

32-bit

0xc000 0000 = 3 GB ; 0x8000 0000 = 2 GB

64-bit

0x0000 0100 0000 0000 = 1 TB (2^40) ;

0x0000 0200 0000 0000 = 2 TB (2^41) << MIPS >>

0x0000 8000 0000 0000 = 128 TB (2^48) << x86_64 4-level >>

Linux Kernel : Memory Management

64-bit VAS

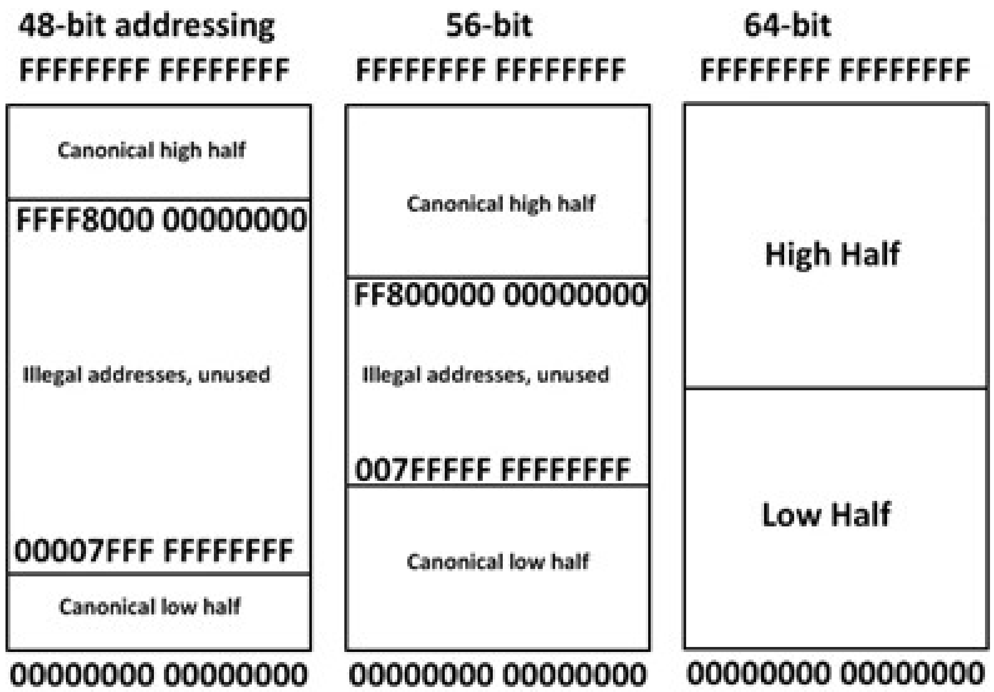


Figure 2 - *Memory Addressing*

Pic: Professional Linux Kernel Architecture, Mauerer

Linux Kernel : Memory Management

Arch-specific VM Splits

With standard 4 KB Page size

Arch	N-Level	Addr Bits	VM "Split"	Userspace Start vaddr	Userspace End vaddr	Kernel-space Start vaddr	Kernel-space End vaddr
IA-32	2	32	3 GB : 1 GB	0x0	0xbfff ffff	0xc000 0000	0xffff ffff
ARM	2	32	2 GB : 2 GB	0x0	0x7fff ffff	0x8000 0000	0xffff ffff
x86_64	4	48	128 TB : 128 TB	0x0	0x0000 7fff ffff ffff	0xffff 8000 0000 0000	0xffff ffff ffff ffff
	5*	56	64 **PB** : 64 **PB**	0x0	0x00ff ffff ffff ffff	0xff00 0000 0000 0000	0xffff ffff ffff ffff
Aarch64	3	39	512 GB : 512 GB	0x0	0x0000 007f ffff ffff	0xffff ff800 0000 000	0xffff ffff ffff ffff
	4	48	256 TB : 256 TB	0x0	0x0000 ffff ffff ffff	0xffff 0000 0000 0000	0xffff ffff ffff ffff

** >= 4.14 Linux*

IMP Update! Please see the newly updated (as of Linux 5.0) x86_64 process memory layout doc here:
https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

03 Jan 2018 : Meltdown and Spectre hardware errors

- **Root cause: CPU speculative code/data fetches**
- **Security concerns**
- **"Due to a processor-level bug (to do with speculative code execution), the traditional approach of keeping kernel paging tables and thus kernel virtual address space (VAS) within the process – anchored in the upper region – is now susceptible to attack! And thus needs to be changed. Linux kernel devs have been working furiously at it…"**
- **Mitigation: KPTI : kernel page table isolation**

```
$ uname -a
Linux ... 4.14.11-300.fc27.x86_64 #1 SMP Wed Jan 3 13:52:28 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
$ dmesg |grep "isolation"
Kernel/User page tables isolation: enabled
$ dmesg |grep -i spectre
kern  :info  : [Sat Mar  3 15:16:00 2018] Spectre V2 : Mitigation: Full generic retpoline
kern  :info  : [Sat Mar  3 15:16:00 2018] Spectre V2 : Spectre v2 mitigation: Filling RSB on context switch
kern  :info  : [Sat Mar  3 15:16:00 2018] Spectre V2 : Spectre v2 mitigation: Enabling Indirect Branch Prediction Barrier
$
```

<< See 2LinuxMM doc for more details >>

Linux Kernel : Memory Management

Meltdown/Spectre Effects

- Traditionally, Linux would keep the kernel paging tables as part of *every process – in it's "upper region" - the 'canonical higher half'*
- With Meltdown/Spectre, that decision has been reversed!
- Performance impact: resulted in (much) higher context-switching times for user ↔ kernel system call code paths
  - Already visible in 'cloud' virtualization workloads, etc
- [Details: "… In current kernels, each process has a single PGD; one of the first steps taken in the KPTI patch series is to create a second PGD. The original remains in use when the kernel is running; it maps the full address space. The second is made active (at the end of the patch series) when the process is running in user space. It points to the same directory hierarchy for pages belonging to the process itself, but the portion describing kernel space (which sits at the high end of the virtual address space) is mostly absent. ..." : *LWN, 20Dec2017, Corbet*
- *New! 14 May 2019: Intel and other hardware vendors disclose a similar class of hardware-related (side chanel) security vulnerabilities, collectively called MDS (Microarchitectural Data Sampling); link*

Linux Kernel : Memory Management

Meltdown/Spectre Effects

- Mitigating the detrimental slowdowns due to the page table isolation (KPTI) solution

  - *Use THP (Transparent Huge Pages)*

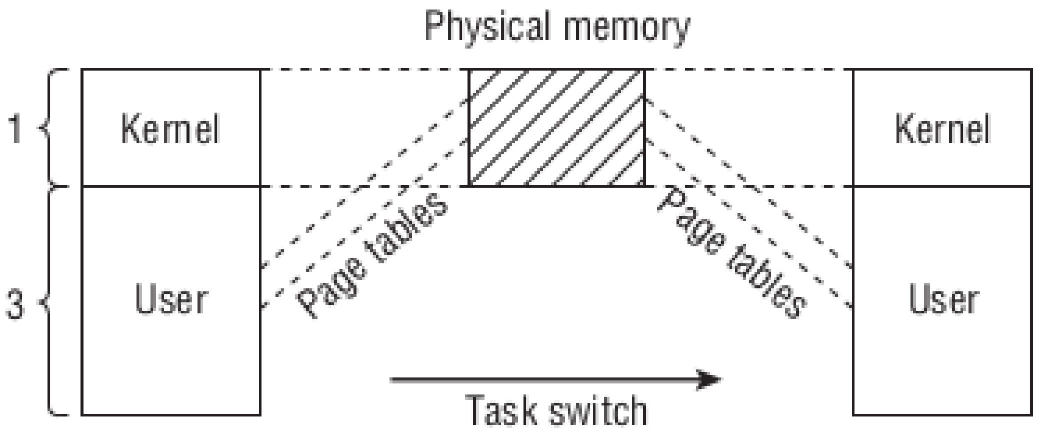
  - *Use PCID (on x86_64); Process Context Identifiers; helps reduce TLB shootdowns significantly. Available "properly" from Linux 4.14*
    V useful article: PCID is now a critical performance/security feature on x86
    [ARM[64] equivalent is the Context ID register; see this: 'ARM Context ID Register & Process Context Switch']

  - *Identify and reduce*
    - *large system call usage*
    - *interrupt sources*

    - *[29Oct2019]*
      Running on Intel? If you want security, disable hyper-threading, says Linux kernel maintainer; Speculative execution bugs will be with us for a very long time

Linux Kernel : Memory Management

- Physical – platform – RAM is "direct-mapped" into the kernel segment at boot
- The OS is the resource manager; it manages RAM, does *not* hoard it!
- Gives it to userspace on demand (via demand paging)



Pic: Professional Linux Kernel Architecture, Mauerer

Figure 3-14: Connection between virtual and physical address space on IA-32 processors.

Linux Kernel : Memory Management

THP – Transparent Huge Pages

A quick summarization

- 2.6.38 onwards
- Kernel address space is always THP-enabled, reducing TLB pressure
  (for eg.: with typical 4Kb page size, to translate (va to pa) 2 MB space,
  one would require (2*1024Kb/4Kb) = 512 TLB entries/translations.
  With THP enabled, each kernel page is 2 MB, thus requiring 1 TLB entry only!)
- Transparent to applications*
- (Userland) Pages are swappable (split into 4k pages & swapped)
- Some CPU overhead (khugepaged checks continously for smaller 4K pages that can be merged together to create a 2MB huge page)
- Currently works only on anonymous memory regions (planned to add support for page cache and tmpfs pages)

Linux Kernel : Memory Management

Show and run the vm_user.c app

Running on a 32-bit IA-32 system (and OS) with a 3 GB : 1 GB VM Split

```
$ getconf -a |grep LONG_BIT
LONG_BIT                           32
$./vm_user
wordsize : 32
&main = 0x0804847d &g = 0x0804a028 &u = 0804a030 &heapptr = 0x082a5008 &loc = 0xbfa6e5d8
$
```

Running on an ARM-32 system (and OS) with a 2 GB : 2 GB VM Split
(Qemu-based ARM Versatile Express platform)

```
$ ./vm_user
32-bit: &main = 0x00008578 &g = 0x00010854 &u = 0x0001085c &heapptr = 0x00011008 &loc = 0x7ebf4d08
$
```

Running on a 64-bit (x86_64) system (and OS) [128 TB : 128 TB VM split]

```
$ getconf -a |grep LONG_BIT
LONG_BIT                           64
$ ./vm_user
wordsize : 64
&main = 0x0000557de98f3165 &g = 0x0000557de98f6010 &u = 0x0000557de98f6018 &heapptr = 0x0000557deb690260 &loc =
0x00007ffcf811766c
$
```

Running on an ARM64 (Raspberry Pi 3) [256 TB : 256 TB VM split]

```
$ ./vm_user
wordsize : 64
&main = 0x0000aaaac50028a4 &g = 0x0000aaaac5013010 &u = 0x0000aaaac5013018 &heapptr = 0x0000aaaac8ec0260 &loc =
0x0000ffffe6e3c1fc
$
```

Linux Kernel : Memory Management

Content

Physical Memory Organization
- [N]UMA, zones
- Buddy Sys Alloc
- Slab Cache

Kernel dynamic allocation APIs
- BSA
- kmalloc
- vmalloc
- Custom slab

Kernel segment – diagrams + dmesg @boot k layout

Page Cache -diagram, blog article [mmap]

OOM killer

Demand paging
- VM overcommit
- Page fault handling basics
- Glibc malloc behavior
- User VAS mapping - proc

Linux Kernel : Memory Management

Physical Memory Organization

SMP – Symmetric Multi Processing

One OS, one RAM, multiple CPUs (cores)

AMP – Asymmetric Multi Processing

> 1 OS, one RAM, multiple CPUs (cores)

UMA – Uniform Memory Access

RAM treated as a uniform hardware resource by the OS; does not matter which CPU a thread is running upon, memory looks the same

NUMA – Non-Uniform Memory Access

RAM “banks” treated as distinct hardware resources by the OS – called ‘nodes’; does matter on which CPU a thread is running upon, should use RAM from the “closest” node.

Linux (Windows, UNIXes) is NUMA-aware.

Linux Kernel : Memory Management

[N]UMA

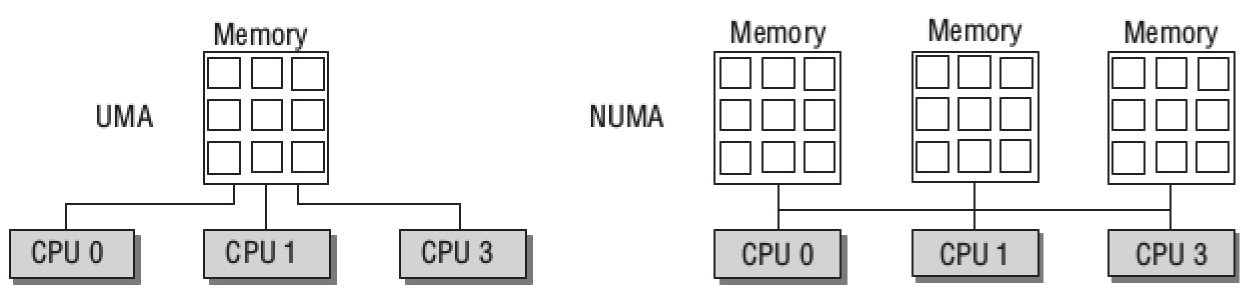


Figure 3-1: UMA and NUMA systems.

Above pic: “Professional Linux Kernel Architecture”, W Mauerer, Wrox Press

NUMA on Wikipedia

NUMA machines are always multiprocessor; each CPU has local RAM available to it to support very fast access; also, all RAM is linked to all CPUs via a bus.
(Above) Image Source

Linux Kernel : Memory Management

NUMA system, 4 nodes

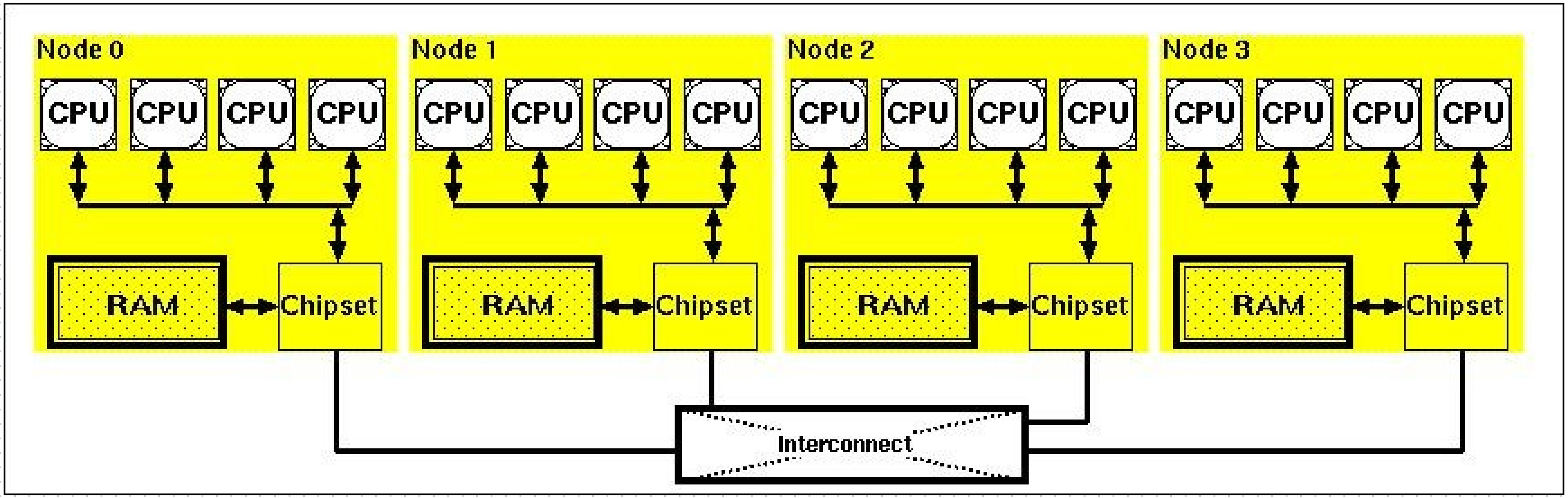
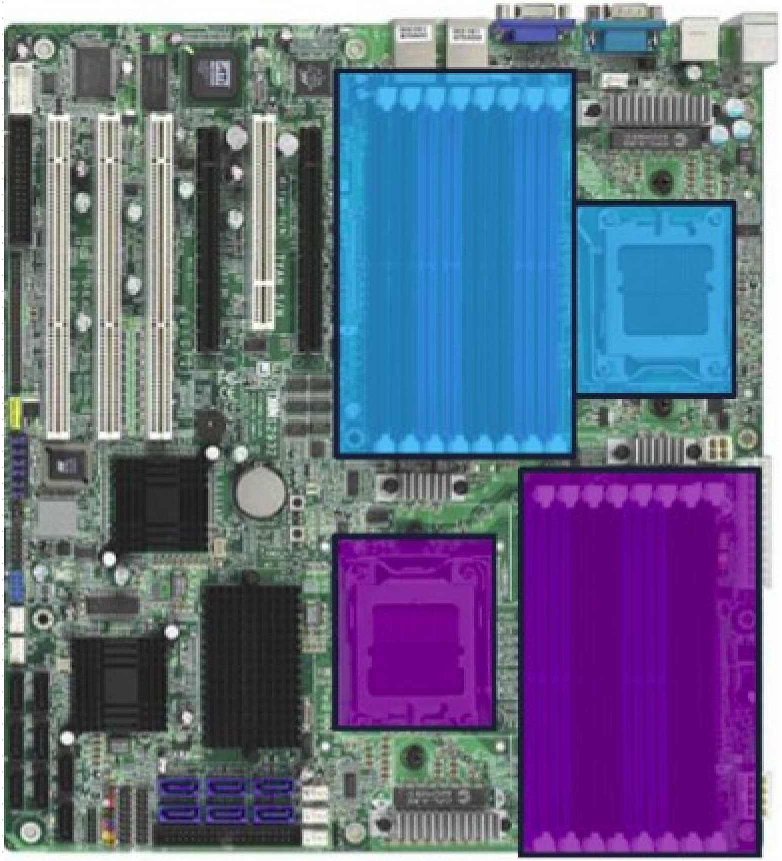


Image Source

Linux Kernel : Memory Management

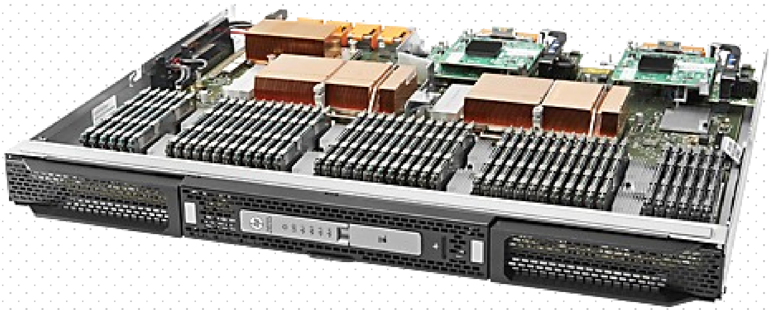
NUMA server



Linux Kernel : Memory Management

Actual NUMA servers

NUMA-Q, IBM

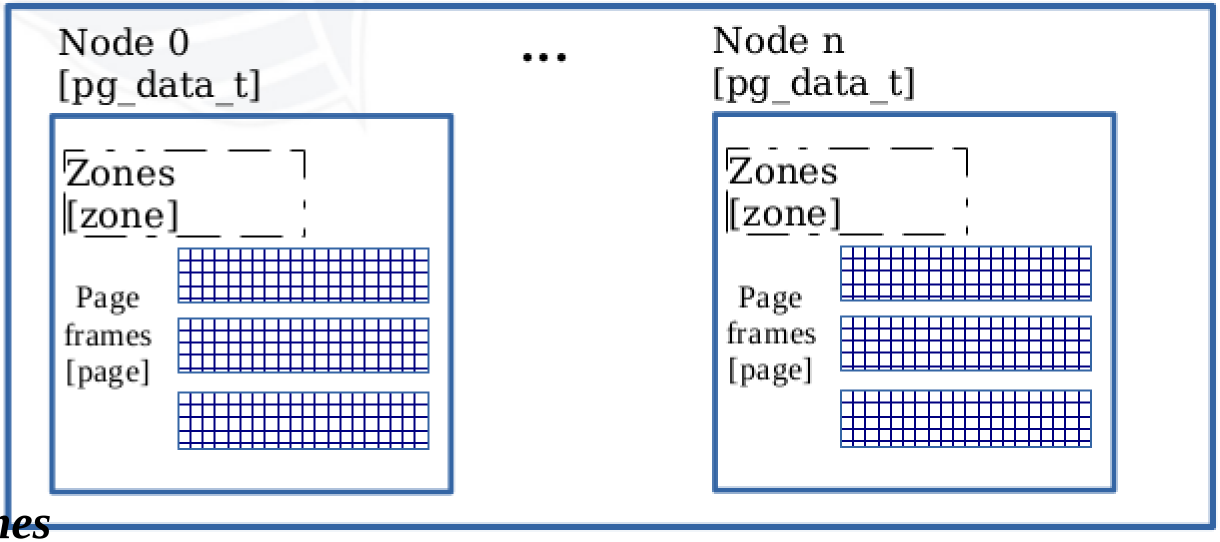


Integrity SuperdomeX, HP

Linux Kernel : Memory Management

Linux

- **RAM is divided into *nodes***
  - One node for each CPU core that has *local RAM* available to it
    - A node is represented by the data structure *pg_data_t*
- **Each node is further split into (at least one, upto four) *zones***
  - ZONE_DMA
  - ZONE_DMA32
  - ZONE_NORMAL
  - ZONE_HIGHMEM



- **Each zone consists of *page frames***
  - A page frame is represented (and managed) by the data structure *page*

Linux Kernel : Memory Management

Emulating a NUMA box with Qemu

```
$ qemu-system-x86_64 --enable-kvm
-kernel <...>/4.4.21-x86-tags/arch/x86/boot/bzImage
-drive file=images/wheezy.img,if=virtio,format=raw
-append root=/dev/vda -m 1024 -device virtio-serial-pci
-smp cores=4 -numa node,cpus=0-1,mem=512M
-numa node,cpus=2-3,mem=512M
-monitor stdio
…
```

(qemu) info numa
2 nodes
node 0 cpus: 0 1
node 0 size: 512 MB
node 1 cpus: 2 3
node 1 size: 512 MB

Zones

- Often enable a workaround over a hardware or software issue. Eg.
  - ZONE_DMA : 0 – 16 MB on old Intel with the ISA bus
  - ZONE_HIGHMEM : what if the 32-bit platform has more RAM than there is kernel address space? (eg. an embedded system with 3 GB RAM on a machine with a 3:1 VM split)
  - … and so on
- The ‘lowmem’ region – direct mapped platform RAM – is often placed as ZONE_NORMAL (but not always)
- Kernel inits zones at boot

False Sharing

“Avoiding and Identifying False Sharing Among Threads”, Intel

In symmetric multiprocessor (SMP) systems, each processor has a local cache.
The memory system must guarantee cache coherence. False sharing occurs whe
threads on different processors modify variables that reside on the same cache li
This invalidates the cache line and forces an update, which hurts performance.
This article covers methods to detect and correct false sharing.

Prevented by ensuring the variables cannot “live” in the same CPU cacheline

An example of false-sharing corrected

Each zone is represented by struct zone, which is defined in the
File : [3.10.24]: include/linux/mmzone.h

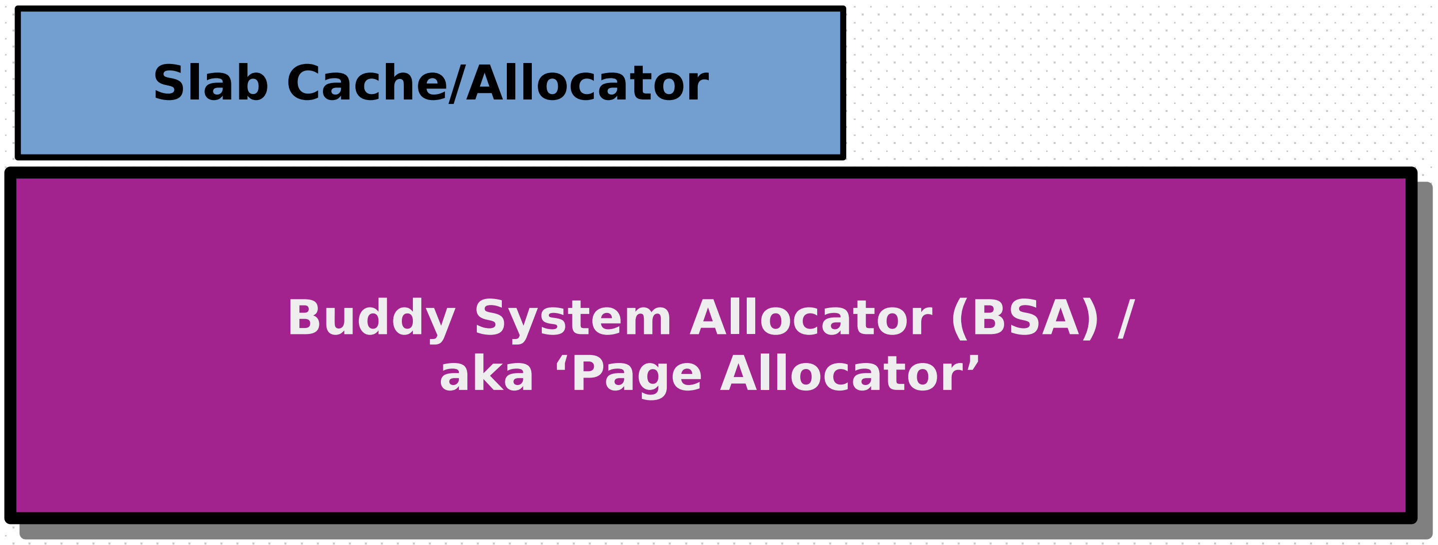
```
struct zone {
[…]
       spinlock_t              lock;
--snip--
        struct free_area        free_area[MAX_ORDER];
--snip--
      ZONE_PADDING(_pad1_)

       /* Fields commonly accessed by the page reclaim scanner */
        spinlock_t              lru_lock;
<<
 File : [3.10.24] : include/linux/mmzone.h
   /*
    * zone->lock and zone->lru_lock are two of the hottest locks in the kernel.
    * So add a wild amount of padding here to ensure that they fall into separate
    * cachelines.  There are very few zone structures in the machine, so space
    * consumption is not a concern here.
    */
>>
...
```

Linux Kernel : Memory Management

Linux Kernel Memory Allocation

- Layered Approach



Buddy System Allocator (BSA)

The kernel's memory alloc/de-alloc "engine"

The key data structure is the 'free list':

- An array of pointers to doubly linked circular lists
- 0 to MAX_ORDER-1 (MAX_ORDER is usually 10); thus 11 entries
- each index is called the 'order' of the list
    - holds a chain of free memory blocks
    - each memory block
        - is of size 2^order
        - is guaranteed to be *physically contiguous*

- *allocations are always rounded power-of-2 pages*
    - exception: when using the *alloc_pages_exact* API

Linux Kernel : Memory Management

The BSA (Buddy System Allocator) free list

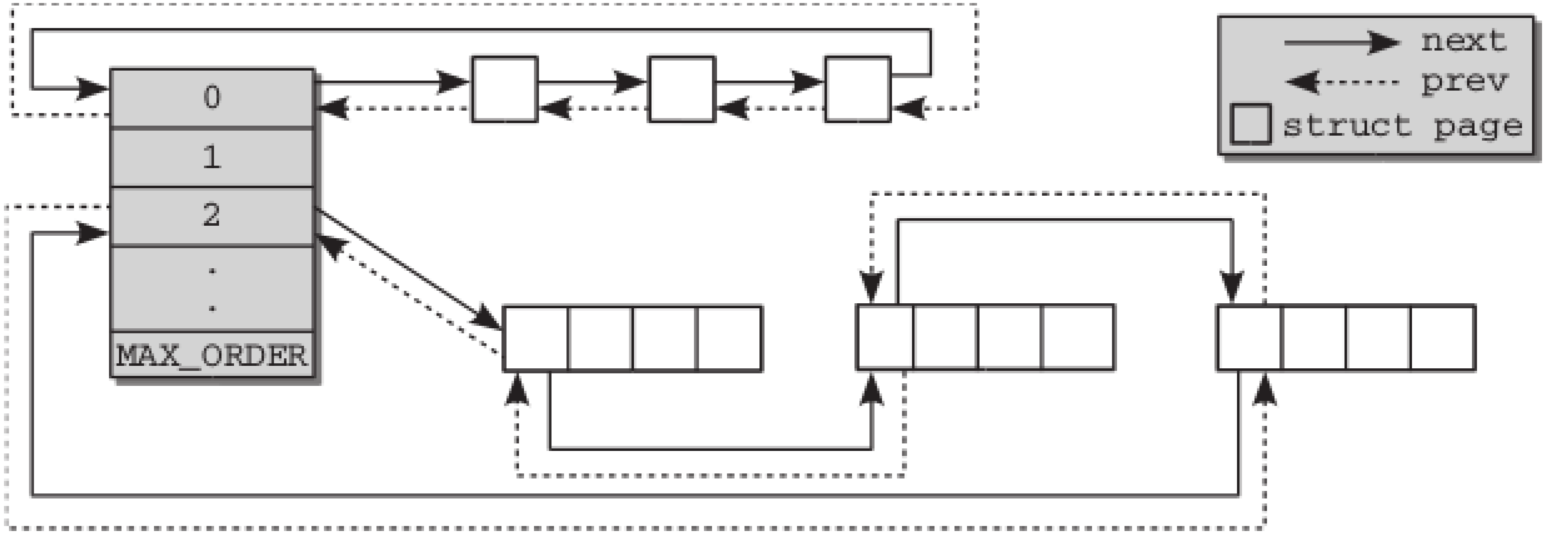


Figure 3-22: Linking blocks in the buddy system.

Linux Kernel : Memory Management

On a system with 1 GB RAM:

```
$ cat /proc/buddyinfo
Node 0, zone      DMA      0      0      0      1      2      1      1      0      1      1      3
Node 0, zone    DMA32     49     32     11      5      4     41     15     13     16     15     90
$
```

On a system with 16 GB RAM:

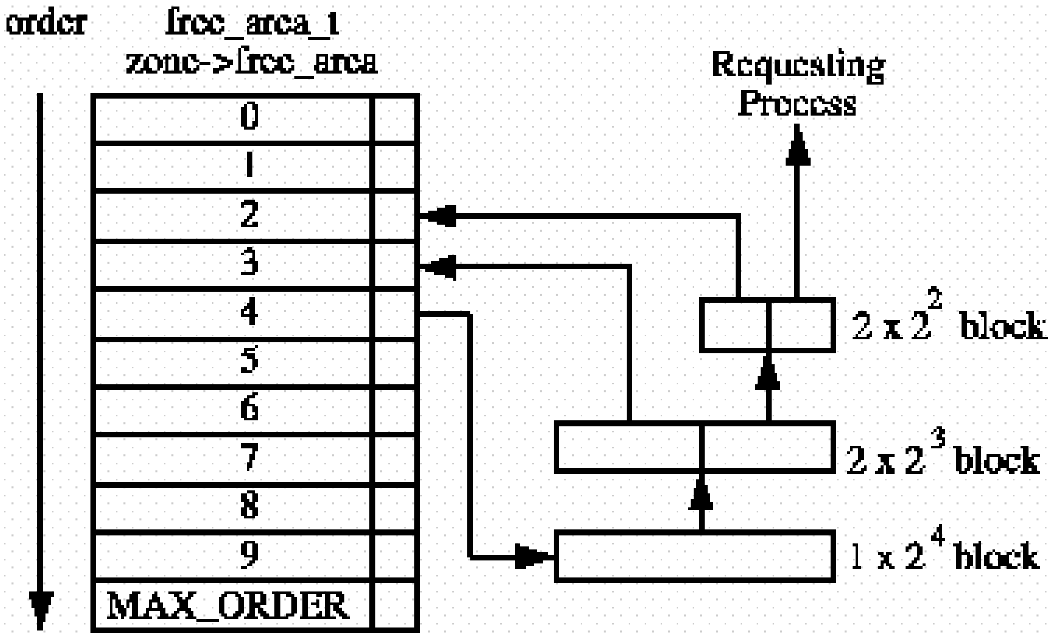
```
$ cat /proc/buddyinfo
Node 0, zone      DMA      2      3      6      2      3      1      0      0      1      1      3
Node 0, zone    DMA32  14617   3480   1303    465    388     97     63     23     11      3      0
Node 0, zone   Normal   1208   1909    303    609    552    180     49     10      3      0      0
$
```

Order 0

Order 10

Linux Kernel : Memory Management

BSA: alloc of 4 KB (2^2) chunk by splitting:
- a 2^4 (16 KB) chunk into 2 chunks of 2^3 (8K) each
- further splitting the 2^3 (8 KB) chunk into 2 chunks of 2^2 (4 KB) each
- the left-over 2^3 chunk goes onto order 3
- the left-over 2^2 chunk goes onto order 2



Linux Kernel : Memory Management

BSA: one freelist per node per zone (in each node)

- Makes the (de)alloc NUMA-aware
- Actually, from 2.6.24, it's more complex: the addition of *per-migration-type* freelists means there are upto six freelists per node per zone *in each node)!

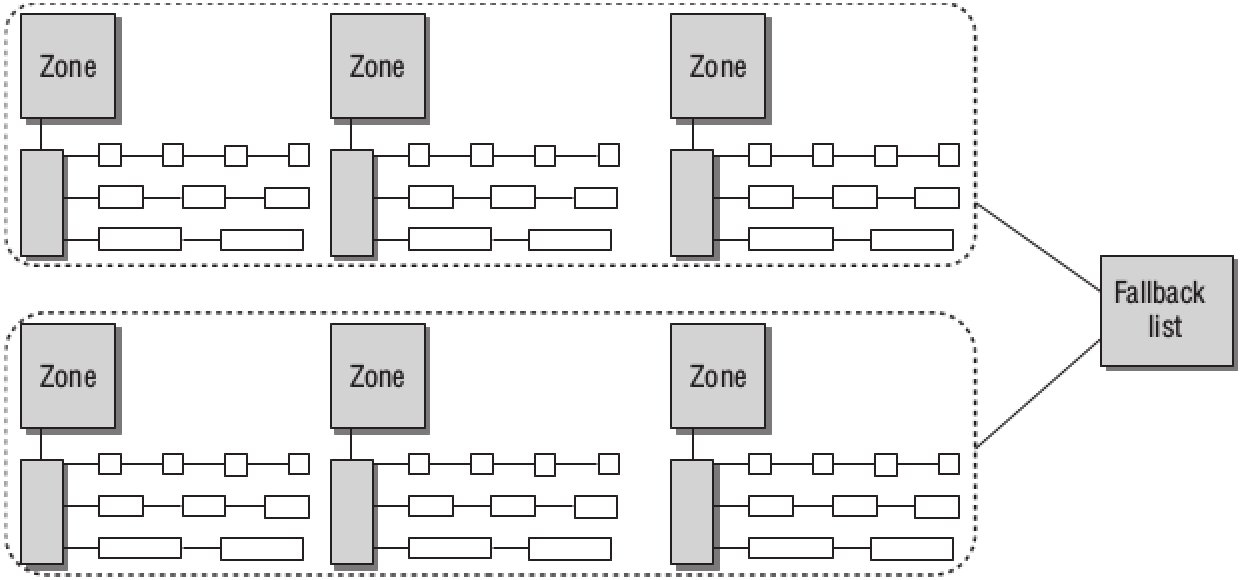


Figure 3-23: Relationship between buddy system and memory zones/nodes.

Linux Kernel : Memory Management

BSA / Page Allocator

Pros

- very fast
- physically contiguous page-aligned memory chunks
- actually helps defragment RAM by *merging* ‘buddy’ blocks
    - buddy block: block of the same size and physically contiguous

Cons

- primary issue: internal fragmentation / **wastage!**
    - granularity is a page; so asking for 128 bytes gets you 4096

Slab Cache / Slab Allocator

- The BSA’s major wastage problem is addressed by the slab allocator

- Slab : two Big Ideas
  - caches ‘objects’ - commonly used kernel data structures
  - caches fragments of RAM

Look it up!

```
sudo vmstat -m
```

Linux Kernel : Memory Management

Check out the available slabs for the kmalloc:

```
                            Num   Total  Size   Pages
$ sudo vmstat -m |grep "^kmalloc\-"
kmalloc-8192                  52     52  8192       4
kmalloc-4096                 109    136  4096       8
kmalloc-2048                 267    272  2048       8
kmalloc-1024                 860   1040  1024       8
kmalloc-512                  547    672   512       8
kmalloc-256                 1198   1728   256      16
kmalloc-192                 1470   1470   192      21
kmalloc-128                 1280   1280   128      32
kmalloc-96                  1890   1890    96      42
kmalloc-64                  3136   3136    64      64
kmalloc-32                  3456   3456    32     128
kmalloc-16                  2816   2816    16     256
kmalloc-8                   3584   3584     8     512
$
```

num: Number of currently active objects
total: Total number of available objects
size: Size of each object
pages: Number of pages with at least one active object

Linux Kernel : Memory Management

Kernel Dynamic Memory Alloc APIs

- BSA (Buddy System Allocator)

- Slab
    - builtin APIs
    - custom slab cache

Kernel Dynamic Memory Alloc APIs

BSA - ‘low-level’ APIs

Most important of them is:

```
unsigned long __get_free_page(gfp_mask);
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);
unsigned long get_zeroed_page(gfp_t gfp_mask);
void *alloc_pages_exact(size_t size, gfp_t gfp_mask);
```

Free with

```
free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned int order);
void free_pages_exact(void *virt, size_t size);
```

Linux Kernel : Memory Management

Kernel Dynamic Memory Alloc APIs

A 'golden rule' :: **'cannot sleep in atomic context'**; hence, cannot call any API that results in a call to *schedule()* in atomic context (any kind of interrupt code is an atomic context: ISR top half, bottom halves (tasklets, softirqs))

GFP (get_free_page) Mask (gfp_t):
- a bitmask of several possible values
- most are used internally
- 'module authors' (driver devs) use these:

GFP_KERNEL : use when it's safe to sleep - process-context *only,* no locks held; might cause the process context to block

GFP_ATOMIC : use when one *cannot* sleep; interrupt-context; high priority; will succeed or fail immediately (no blocking)

__GFP_ZERO : zeroes out the memory region

Linux Kernel : Memory Management

Kernel Dynamic Memory Alloc APIs

Which Flag to Use When

Situation	*Solution*
Process context, can sleep	Use GFP_KERNEL
Process context, cannot sleep	Use GFP_ATOMIC, or perform your allocations with GFP_KERNEL at an earlier or later point when you can sleep
Interrupt handler	Use GFP_ATOMIC
Softirq	Use GFP_ATOMIC
Tasklet	Use GFP_ATOMIC

Linux Kernel : Memory Management

Example *Low-level BSA*

drivers/hv/connection.c

```
...
 vmbus_connection.int_page =
   (void *)__get_free_pages(GFP_KERNEL|__GFP_ZERO, 0);
         if (vmbus_connection.int_page == NULL) {
                 ret = -ENOMEM;
                 goto cleanup;
         }
...
 free_pages((unsigned long)vmbus_connection.int_page, 0);
 vmbus_connection.int_page = NULL;
...
```

Linux Kernel : Memory Management

Slab Allocator / Slab Cache APIs

- Layered above the BSA
- Gets *it's* memory from the BSA

APIs ::
kmalloc() or kzalloc()
- Vast majority of kernel allocations
- Meant for the usage case where the amount of memory required is less than a page; this is the common case!!

```
#include <linux/slab.h>
void *kmalloc(size_t size, gfp_t flags);
void *kzalloc(size_t size, gfp_t flags);
```

Free the memory with:

```
void kfree(const void *objp);  // for both k[m|z]alloc()
void kzfree(const void *objp); // zero the mem & free it
                                  (security)
```

Linux Kernel : Memory Management

Example *Slab allocator*

drivers/hv/connection.c

```
...
msginfo = kzalloc(sizeof(*msginfo) +
             sizeof(struct vmbus_channel_initiate_contact),
             GFP_KERNEL);
if (msginfo == NULL) {
         ret = -ENOMEM;
         goto cleanup;
}
...

kfree(msginfo);
```

An unwritten rule: GFP_KERNEL allocations for low-order (<=PAGE_ALLOC_COSTLY_ORDER (=3)) never fail (implies, <= 8 pages)!

Runtime Usage of kmalloc

One can actually "track" these functions via the powerful ***Kprobes*** kernel framework! Below, outout seen while using a **Jprobe and a Kretprobe on `__kmalloc`** :

```
[...]

 jp_kmalloc: 40 callbacks suppressed
  __kmalloc(96)
 PRINT_CTX:: [000] systemd-journal :215   | d..0
  __kmalloc(24)
 PRINT_CTX:: [000][kworker/u2:2]:3529   | d..0
  __kmalloc(152)
 ret_handler: 40 callbacks suppressed
   = 0xffff95e9b8a3db40
 PRINT_CTX:: [000][kthreadd]:2   | d..0
  __kmalloc(32)
   = 0xffff95e9b8a457e0

[...]
```

Note- Jprobes removed from 4.15 onward

k[m|z]alloc Upper Limit on a Single Allocation

- From 2.6.22, the upper limit on x86, ARM, x86_64, etc is **4 MB**

- Technically, a function of the CPU page size and the value of MAX_ORDER
    - with a page size of 4 KB and MAX_ORDER of 10, the upper limit is 4 MB

A detailed article describing the same:
[kmalloc and vmalloc : Linux kernel memory allocation API Limits]

kmalloc Pros

- Uses the kernel’s identity-mapped RAM (obtained and mapped at boot)
  - usually in ZONE_NORMAL
  - called the **‘lowmem’** region
- it’s thus **very fast** (pre-mapped, no page table setup required)

- returned memory chunks are guaranteed to be
  - *physically* contiguous
  - on a hardware cacheline boundary

kmalloc Limitations
- there is an upper limit
- security: by default, freed memory is not cleared, which could result in info-leakage scenarios. Can build the kernel with CONFIG_PAGE_POISONING (performance impact)

Linux Kernel : Memory Management

The *ksize* API, given a pointer to slab-allocated memory (in effect, memory allocated via the *kmalloc*), returns the actual number of bytes allocated.

```
size_t ksize(const void *objp);
```

It could be more than what was requested. F.e.

```
size_t actual=0;
void *ptr = kmalloc(160, GFP_KERNEL); // ask for 160 bytes
actual = ksize(ptr);  // will get 192 bytes (actual ← 192)
```

Demo: the mm/allocsize_tests/ksz_test kernel module

Linux Kernel : Memory Management

vmalloc

- When you require more memory than kmalloc can provide (typically 4 MB)

- vmalloc provides a *virtually contiguous* memory region

- max size depends on the hardware platform and the kernel config; the kernel’s VMALLOC region size – a global pool for all vmalloc’s

- only for software use (f.e. not ok for DMA transfers); *only* process context (not in interrupt code); might cause process context to block

- slower (page table setup, demand paging based memory, unlike the kmalloc)

```
void *vmalloc(unsigned long size);
void *vzalloc(unsigned long size);
void vfree(const void *addr);
```

Linux Kernel : Memory Management

vmalloc: example code

fs/cramfs/uncompress.c

```
…
int cramfs_uncompress_init(void)            << see next slide >>
{
        if (!initialized++) {
          stream.workspace = vmalloc(zlib_inflate_workspacesize());
                if (!stream.workspace) {
                        initialized = 0;
                        return -ENOMEM;
                }
...
```

arch/x86/kvm/cpuid.c

```
...
  r = -ENOMEM;
  cpuid_entries = vmalloc(sizeof(struct kvm_cpuid_entry) *
                   cpuid->nent);
  if (!cpuid_entries)
          goto out;
...
```

Examine current vmalloc allocations via procfs (on an ARM)

```
ARM # cat /proc/vmallocinfo
0xa0800000-0xa0802000    8192 of_iomap+0x40/0x48 phys=1e001000 ioremap
0xa0802000-0xa0804000    8192 of_iomap+0x40/0x48 phys=1e000000 ioremap
0xa0804000-0xa0806000    8192 l2x0_of_init+0x68/0x234 phys=1e00a000 ioremap
0xa0806000-0xa0808000    8192 of_iomap+0x40/0x48 phys=10001000 ioremap

...

0xa088b000-0xa088d000    8192 devm_ioremap+0x48/0x7c phys=10016000 ioremap
0xa088d000-0xa0899000   49152 cramfs_uncompress_init+0x38/0x74 pages=11 vmalloc
0xa0899000-0xa08dc000  274432 jffs2_zlib_init+0x20/0x80 pages=66 vmalloc

...
#
```

Also:

When unsure, can use the **kvmalloc()**; will invoke kmalloc() as a first preference, and fallback to the vmalloc():

```
void *kvmalloc(size_t size, gfp_t flags);
```

Linux Kernel : Memory Management

TIP: to get memory with particular protections

mm/vmalloc.c

```
...
 *  For tight control over page level allocator and protection flags
 *  use __vmalloc() instead.
...

void *__vmalloc(unsigned long size, gfp_t gfp_mask, pgprot_t prot)
...
```

arch/x86/include/asm/pgtable_types.h defines these protection symbols (and more):

```
...
PAGE_KERNEL_RO
PAGE_KERNEL_RX
PAGE_KERNEL_EXEC
PAGE_KERNEL_NOCACHE
PAGE_KERNEL_LARGE
PAGE_KERNEL_LARGE_EXEC
PAGE_KERNEL_IO
PAGE_KERNEL_IO_NOCACHE
...
```

Custom Slab Cache

- If a data structure in your kernel code is very often allocated and de-allocated, it's perhaps a good candidate for a ***custom slab cache***
- Slab layer exposes APIs to allow custom cache creation and management

- **APIs**

```
struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
            unsigned long,
            void (*)(void *));
void kmem_cache_destroy(struct kmem_cache *);


void *kmem_cache_[z]alloc(struct kmem_cache *cachep, gfp_t flags);
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

<< Custom Slab Cache code example >>

Custom Slab Cache

Misc

```
unsigned int kmem_cache_size(struct kmem_cache *s);
```

Cache Shrinker Interfaces

```
int kmem_cache_shrink(struct kmem_cache *cachep);
extern int register_shrinker(struct shrinker *);
extern void unregister_shrinker(struct shrinker *);
```

Linux Kernel : Memory Management

Managed memory for drivers with the devres APIs

- Devres : *device resources* manager
- A framework developed for the device model, enabling auto-freeing of memory allocated via te ‘devres’ APIs
- The freeing occurs on driver detach

In place of	*Devres API*
kmalloc / kzalloc	devm_kmalloc / devm_kzalloc
dma_alloc_coherent	dmam_alloc_coherent

Many more ‘managed’ interfaces available; see the kernel documentation here:
https://www.kernel.org/doc/Documentation/driver-model/devres.txt

Linux Kernel : Memory Management

Dynamic Memory Allocation APIs to Use

Required Memory	Method to Use	API(s)
Physically contiguous perfectly rounded power-of-2 pages (of size between 1 page and 4 MB)	BSA / page allocator	`alloc_page[s]`, `__get_free_page[s]`, `get_zeroed_page`, `alloc_pages_exact`
Common case: physically contiguous memory of size less than 1 page	Slab Allocator (cache)	`kmalloc`, `kzalloc`
Physically contiguous memory (not a power-of-2) of size between 1 page and kmalloc max slab (typically 8 KB) or higher (upto 4 MB typically)	Slab Allocator (cache)	kmalloc, kzalloc : but check actual size allocated with ksize() and accordingly optimize
Virtually contiguous memory of large size (> 4 MB typically)	Indirect via BSA	`vmalloc` or CMA
Custom data structures	Custom slab caches	`kmem_cache_*` APIs

Linux Kernel : Memory Management

Kernel Segment

- Layout of the kernel virtual address space
- CPU-dependant
- <see "Kernel Memory Layout on ARM Linux" :
http://www.arm.linux.org.uk/developer/memory.txt

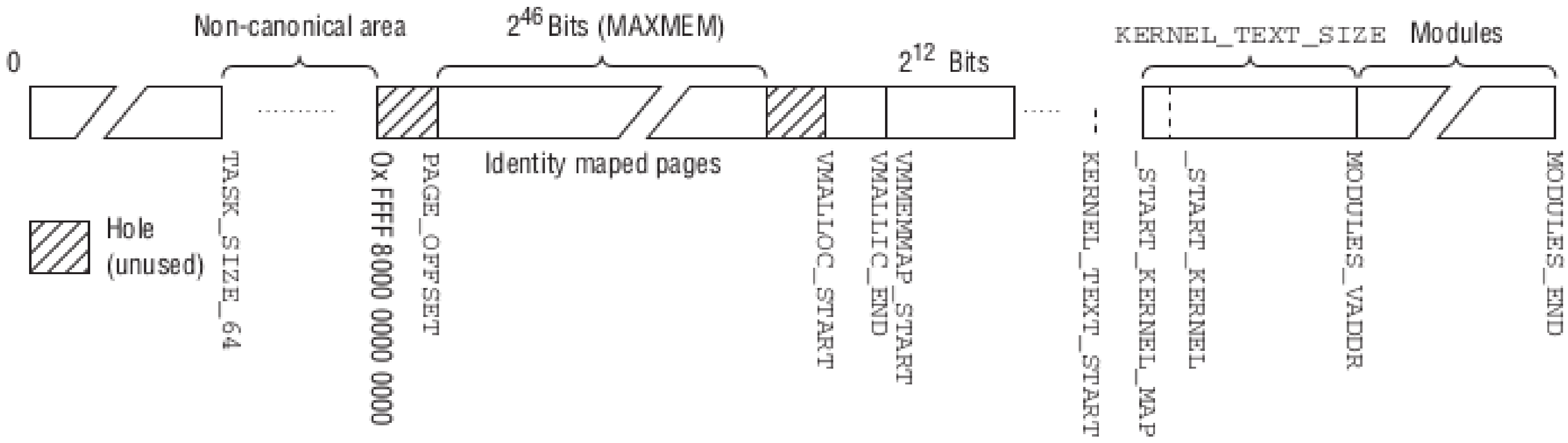
Eg.: On an ARM-32 Linux (Versatile Express CA-9 platform with 512 MB RAM):

```
$ dmesg
[...]
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
    vmalloc : 0xa0800000 - 0xff800000   (1520 MB)
    lowmem  : 0x80000000 - 0xa0000000   ( 512 MB)
    modules : 0x7f000000 - 0x80000000   (  16 MB)
      .text : 0x80008000 - 0x80800000   (8160 kB)
      .init : 0x80b00000 - 0x80c00000   (1024 kB)
      .data : 0x80c00000 - 0x80c664cc   ( 410 kB)
       .bss : 0x80c68000 - 0x814724b8   (8234 kB)
```

Linux Kernel : Memory Management

Kernel Segment on the x86_64

```
#define __PAGE_OFFSET _AC(0xffff810000000000, UL)
#define PAGE_OFFSET __PAGE_OFFSET
#define MAXMEM _AC(0x3fffffffffff, UL)
```



Source: PLKA

Linux Kernel : Memory Management

Kernel Segment on a Yocto-emulated Aarch64 with 512 MB RAM

```
$ dmesg
[...]
[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffffff8000000000 - 0xffffff8008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffffff8008000000 - 0xffffffbebfff0000   (   250 GB)
[    0.000000]       .text : 0xffffff8008080000 - 0xffffff80086c0000   (  6400 KB)
[    0.000000]     .rodata : 0xffffff80086c0000 - 0xffffff8008860000   (  1664 KB)
[    0.000000]       .init : 0xffffff8008860000 - 0xffffff8008900000   (   640 KB)
[    0.000000]       .data : 0xffffff8008900000 - 0xffffff80089bd800   (   758 KB)
[    0.000000]        .bss : 0xffffff80089bd800 - 0xffffff8008a1969c   (   368 KB)
[    0.000000]     fixed   : 0xffffffbefe7fd000 - 0xffffffbefec00000   (  4108 KB)
[    0.000000]     PCI I/O : 0xffffffbefee00000 - 0xffffffbeffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffffffbf00000000 - 0xffffffc000000000   (     4 GB  maximum)
[    0.000000]               0xffffffbf00000000 - 0xffffffbf00800000   (     8 MB actual)
[    0.000000]     memory  : 0xffffffc000000000 - 0xffffffc020000000   (   512 MB)
                                                 << 'lowmem' region – platform RAM >>
[….]
```

Linux Kernel : Memory Management

Kernel Segment on a Raspberry Pi 3 B+ Aarch64 with 1 GB RAM
running 64-bit Ubuntu 18.04.2 LTS

```
$ dmesg
[...]
Virtual kernel memory layout:
modules : 0xffff000000000000 - 0xffff000008000000   (   128 MB)
vmalloc : 0xffff000008000000 - 0xffff7dffbfff0000   (129022 GB)
  .text : 0x        (ptrval) - 0x        (ptrval)   ( 11776 KB)
.rodata : 0x        (ptrval) - 0x        (ptrval)   (  4096 KB)
  .init : 0x        (ptrval) - 0x        (ptrval)   (  6144 KB)
  .data : 0x        (ptrval) - 0x        (ptrval)   (  1343 KB)
   .bss : 0x        (ptrval) - 0x        (ptrval)   (  1003 KB)
fixed   : 0xffff7dfffe7fb000 - 0xffff7dfffec00000   (  4116 KB)
PCI I/O : 0xffff7dfffee00000 - 0xffff7dffffe00000   (    16 MB)
vmemmap : 0xffff7e0000000000 - 0xffff800000000000   (  2048 GB maximum)
          0xffff7e4d36000000 - 0xffff7e4d36ed0000   (    14 MB actual)
memory  : 0xffff934d80000000 - 0xffff934dbb400000   (   948 MB)
                                                  << 'lowmem' region – platform RAM >>
[...]
```

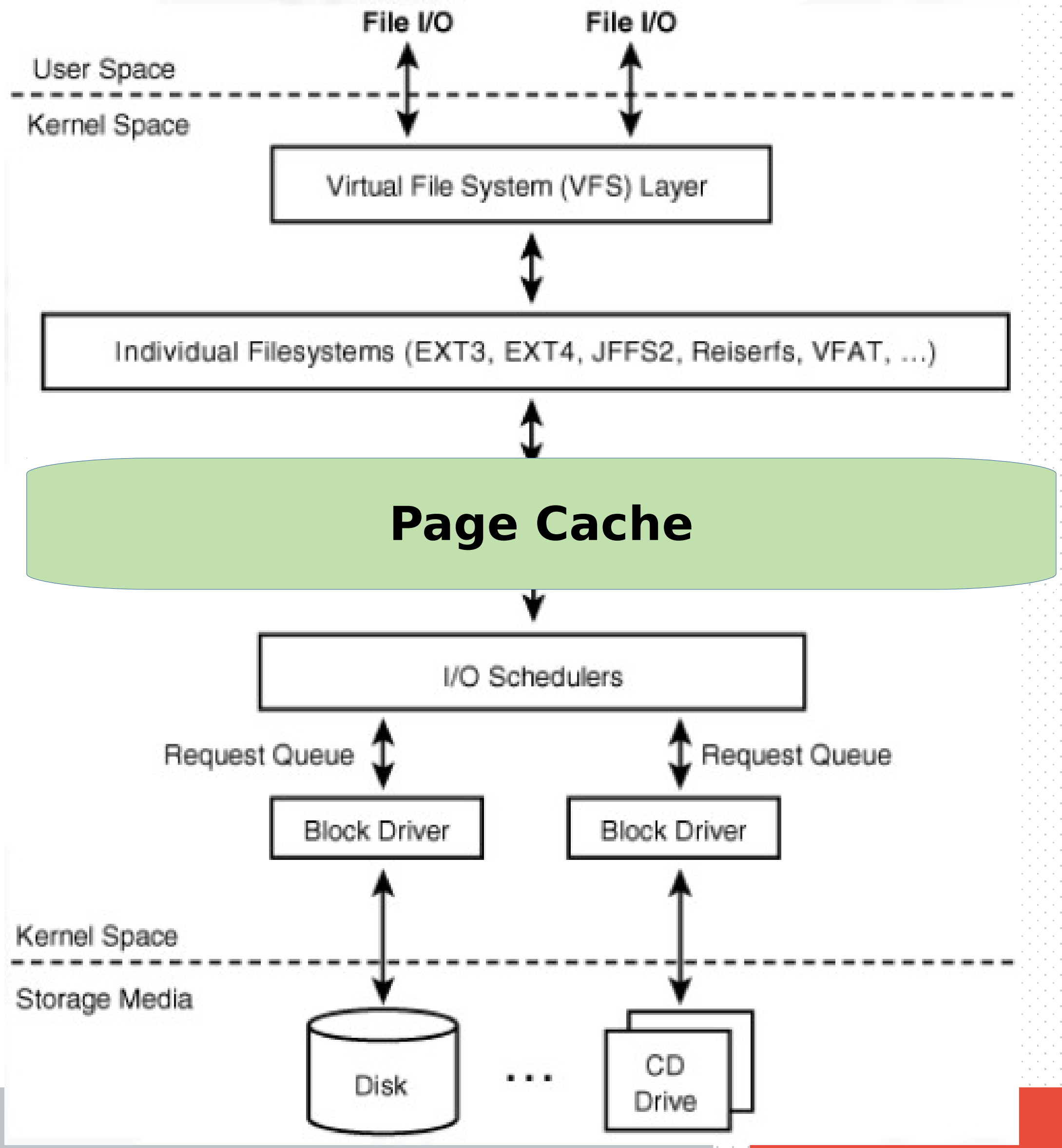
Linux Kernel : Memory Management

The Page Cache

Fundamental property of a cache:

(a) the memory medium of the memory that is being cached is *much* slower to access than the cache

(b) princple of locality of reference: spatial (in space) and temporal (in time)



Page Cache

Bypass the page cache by using O_DIRECT in open(2)

Clean – empty out – the page cache:

`sync && echo 1 > /proc/sys/vm/drop_caches`

```
# free -h
              total    used    free    shared   buff/cache   available
Mem:           15G     9.1G    290M     975M        6.2G        5.3G
Swap:         7.6G       0B    7.6G
Total:         23G     9.1G    7.9G
# sync
# echo 1 > /proc/sys/vm/drop_caches
# free -h
              total    used    free    shared   buff/cache   available
Mem:           15G     9.1G    4.8G      975M       1.7G        5.3G
Swap:         7.6G       0B    7.6G
Total:         23G     9.1G     12G
#
```

Page Cache

The article
“Page Cache , the Affair Between Memory and Files”, Gustav Duarte,
clearly demonstrates why using memory-mapped IO is far superior to the “usual” approach – using the read/write system calls (or library equivalents) in a loop – especially for the use-case where the amount of IO to be performed is quite large.

The OOM (Out Of Memory) Killer and Demand Paging

- Where’s the VM?
  - In the ‘memory pyramid’
  - Registers, CPU Caches, RAM, Swap, (+ newer: nvdimm’s)
- What if – worst case scenario - *all of these are completely full!?*
- *Then the OOM Killer jumps in, and*
  - Kills the process (and descendants) with highest memory usage
  - Uses heuristics to determine the target process

Linux Kernel : Memory Management

The OOM (Out Of Memory) Killer and Demand Paging

Then the OOM Killer jumps in, and

- Kills the process (and descendants) with highest memory usage
- Uses heuristics to determine the target process

- OOM Score (in /proc/<pid>/oom_score)
- Range: [0-1000]
  - 0 : the process is not using any memory available to it
  - 1000 : the process is using 100% of memory available to it

- oom_score_adj
  - Net score = oom_score + oom_score_adj
  - So: oom_score_adj = -1000 => *never kill this*
  - oom_score_adj = +1000 => always *kill this*

- *Details: see the man page on proc(5) – search for oom_**

The OOM (Out Of Memory) Killer and Demand Paging

A script to display the OOM 'score' and 'adjustment' values:

```
# Src: https://dev.to/rrampage/surviving-the-linux-oom-killer-2ki9
printf 'PID\tOOM Score\tOOM Adj\tCommand\n'
while read -r pid comm
do
   [ -f /proc/$pid/oom_score ] && [ $(cat /proc/$pid/oom_score) != 0 ] &&
       printf '%d\t%d\t\t%d\t%s\n' "$pid" "$(cat /proc/$pid/oom_score)" "$(cat
/proc/$pid/oom_score_adj)" "$comm"
done < <(ps -e -o pid= -o comm=) | sort -k 2nr

$ ./show_oom_score
PID       OOM Score    OOM Adj  Command
23584       93         0         VirtualBoxVM
3176       49          0         Web Content
9540       39          0         Web Content
3906       37          0         Web Content
3115       28          0         firefox
3428       23          0         Web Content
[...]
$
```

Linux Kernel : Memory Management

The OOM (Out Of Memory) Killer and Demand Paging

- Can one deliberately invoke the OOM killer?
  - Easy!

    `# echo f > /proc/sysrq-trigger`

  - “Manually”: write a “crazy allocator” ‘C’ program
    - *Check out the **oom_killer_try.c** program now*

Linux Kernel : Memory Management

Demand paging
- VM overcommit
- Page fault handling basics
- Glibc malloc behavior
- User VAS mapping - proc

Linux Kernel : Memory Management