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

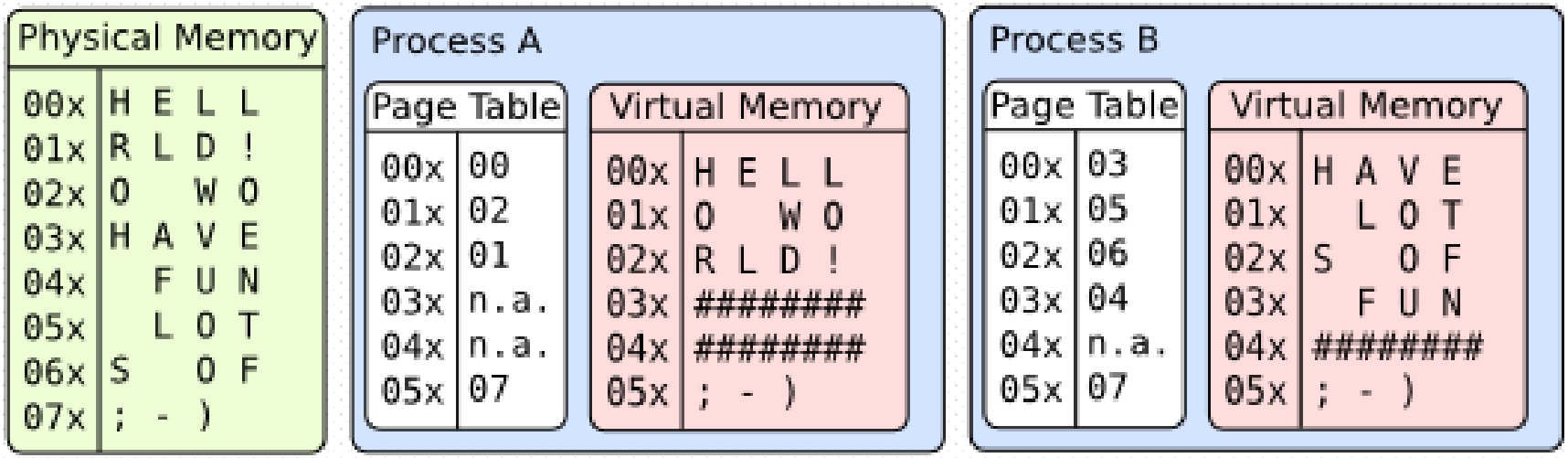
- **Fault Isolation**

  One process crashing will not cause others to be affected

  Multithreaded (firefox) vs multiprocess (chrome)

Linux Kernel : Memory Management

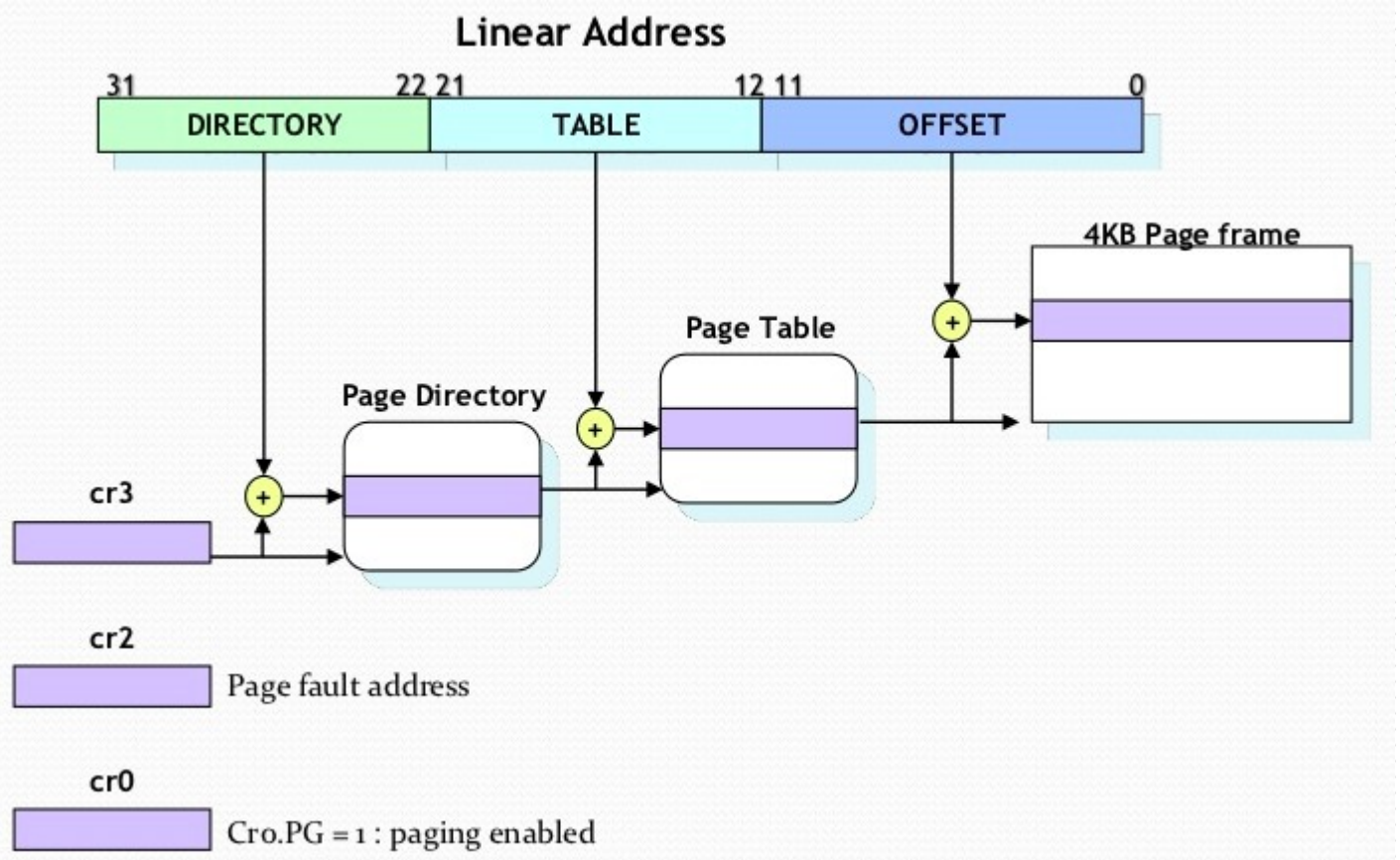
Paging Tables – a *very* simplistic and conceptual view



Source

Actual details on Linux: https://stackoverflow.com/a/36872282/779269

Paging on the x86_32 : 2-level paging



Source

Paging on the x86_64 : 4-level

Source: ULK3

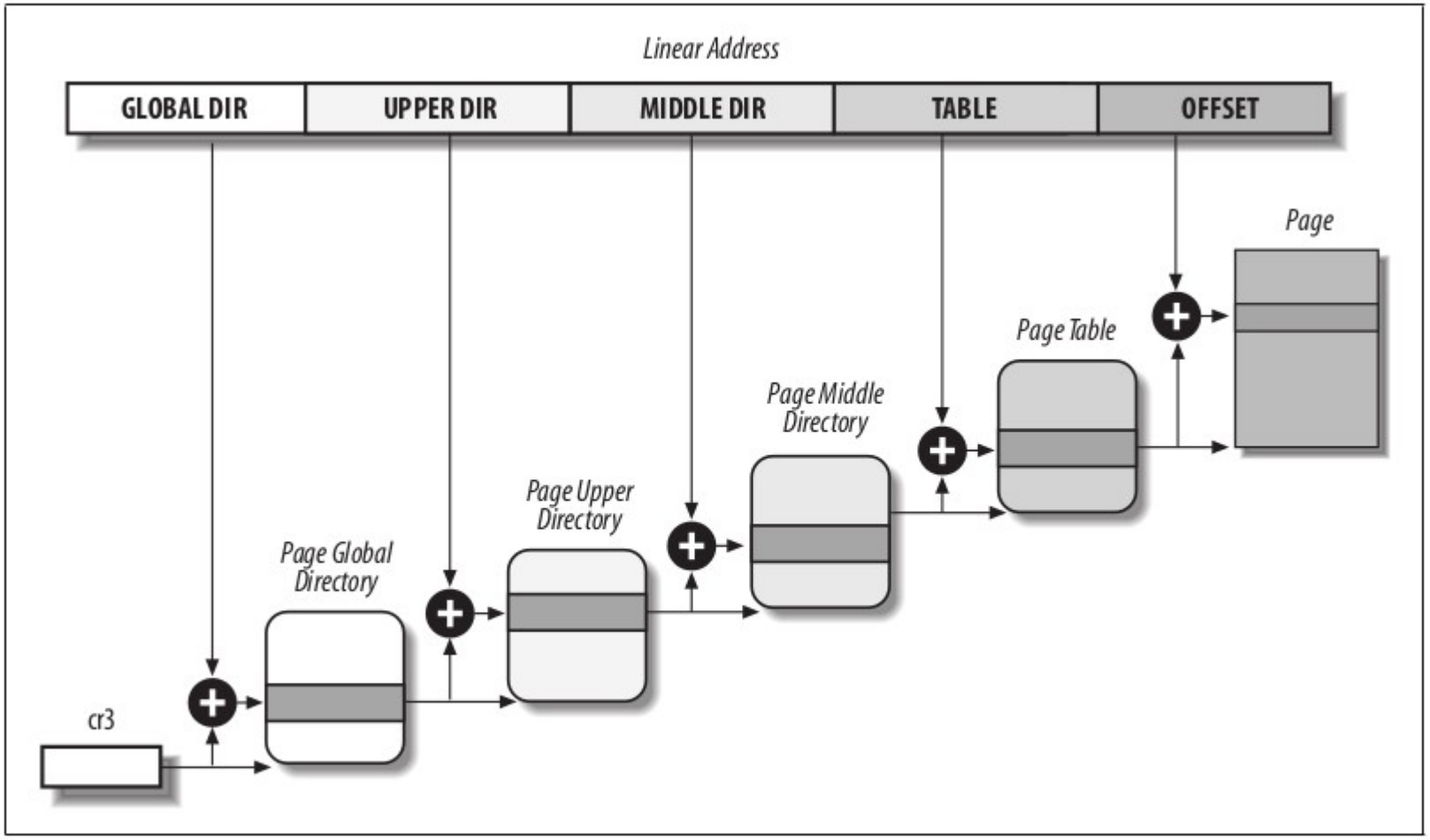
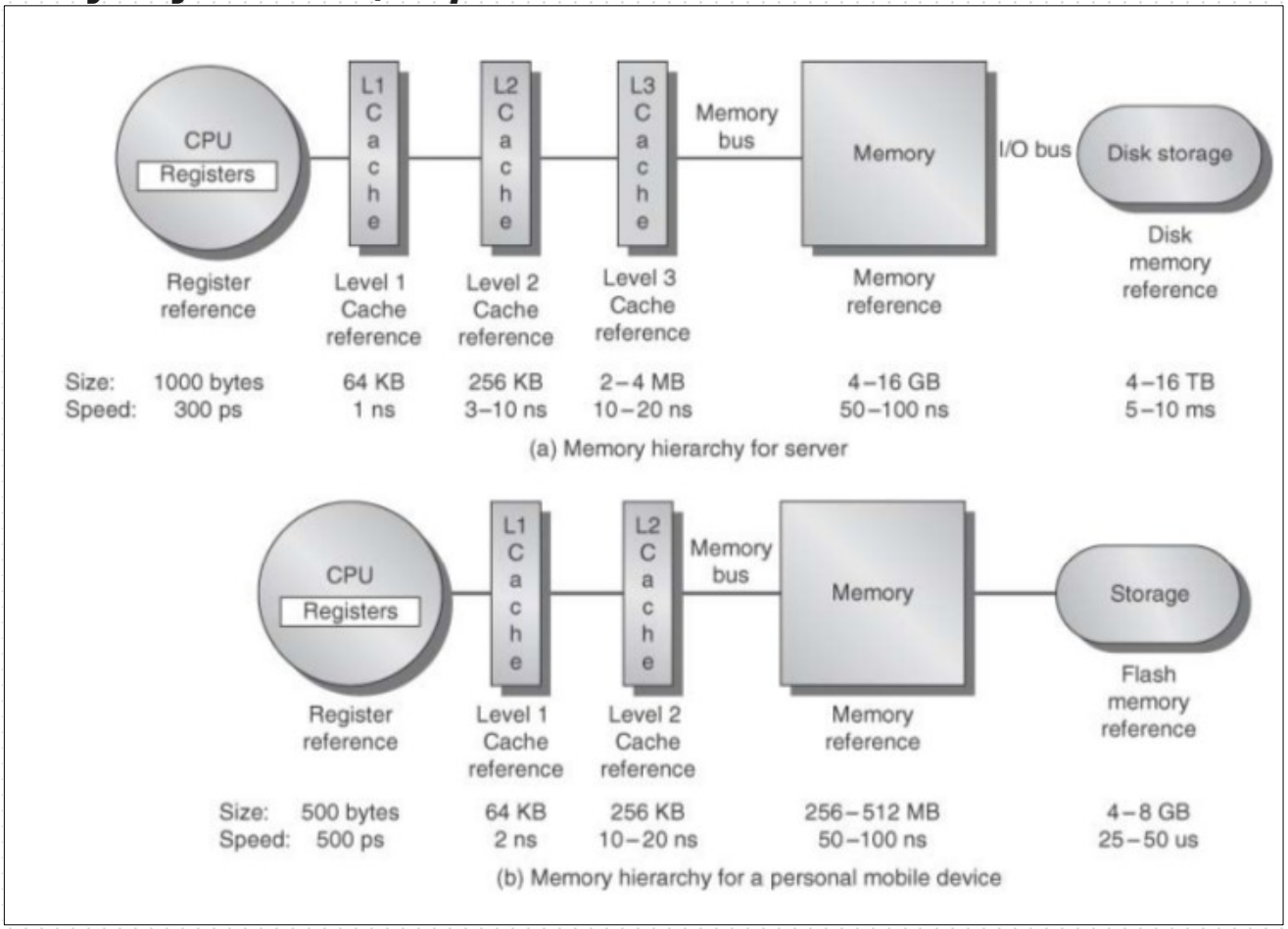


Figure 2-12. The Linux paging model

Linux Kernel : Memory Management

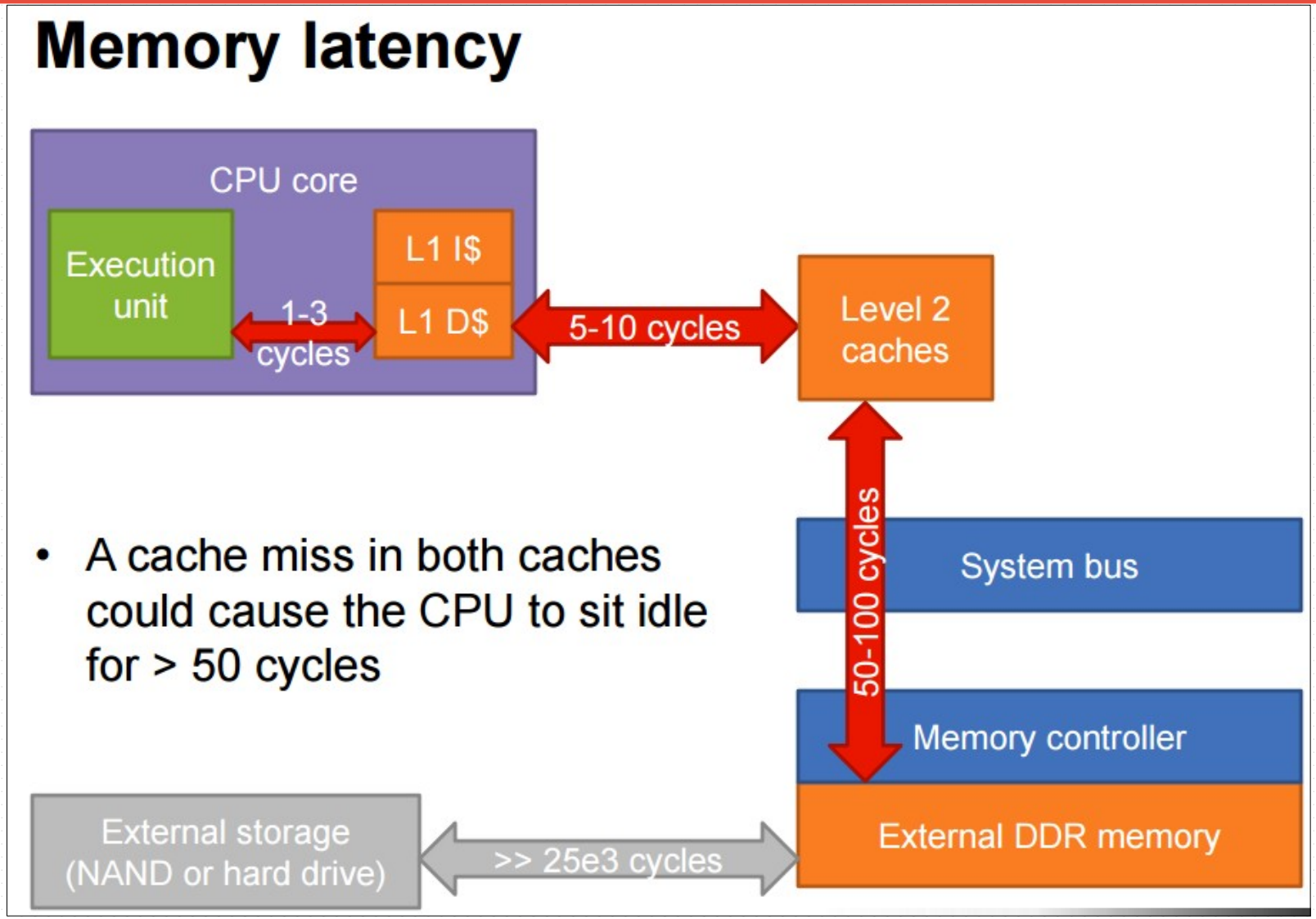
Memory Pyramid ; *Speed* and Size



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

Linux Kernel : Memory Management

CPU Caches



Linux Kernel : Memory Management

CPU Caching : measuring LLC (Last Level Cache) hits/misses

1. With perf !

sudo perf top -e cache-misses -e cache-references -a -ns pid,cpu,comm

2. With the modern approach : eBPF !

$ sudo llcstat-bpfcc

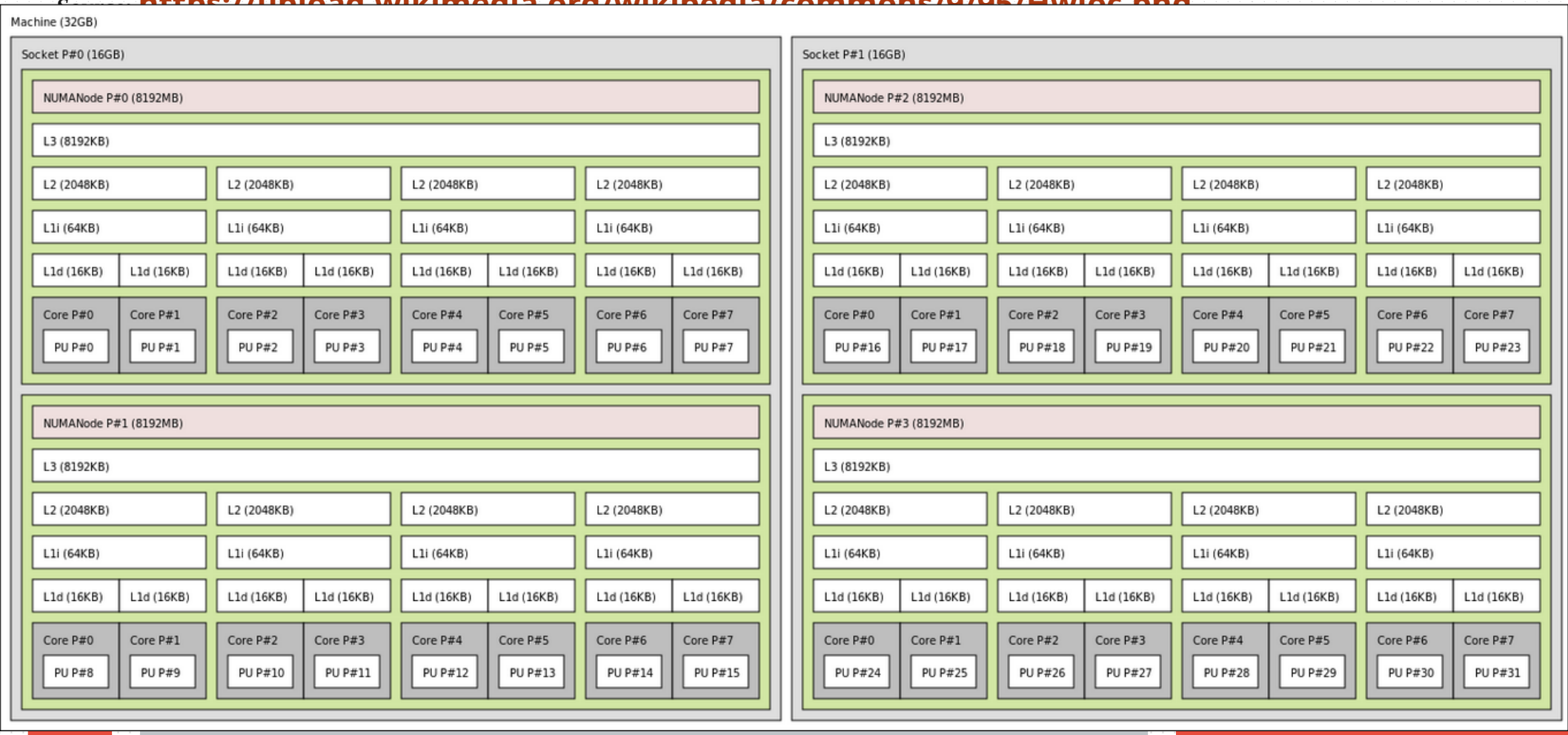
```
Running for 10 seconds or hit Ctrl-C to end.
CPID      NAME             CPU     REFERENCE         MISS    HIT%
3828      Chrome_ChildIOT  3            8300        31400   0.00%
223043    chrome           2            6900        29700   0.00%
129894    nspr-3           3            7500         6100  18.67%
0         swapper/0        0         1432900       322500  77.49%
283       jbd2/sda6-8      0            2500         6200   0.00%
[...]
```

Linux Kernel : Memory Management

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

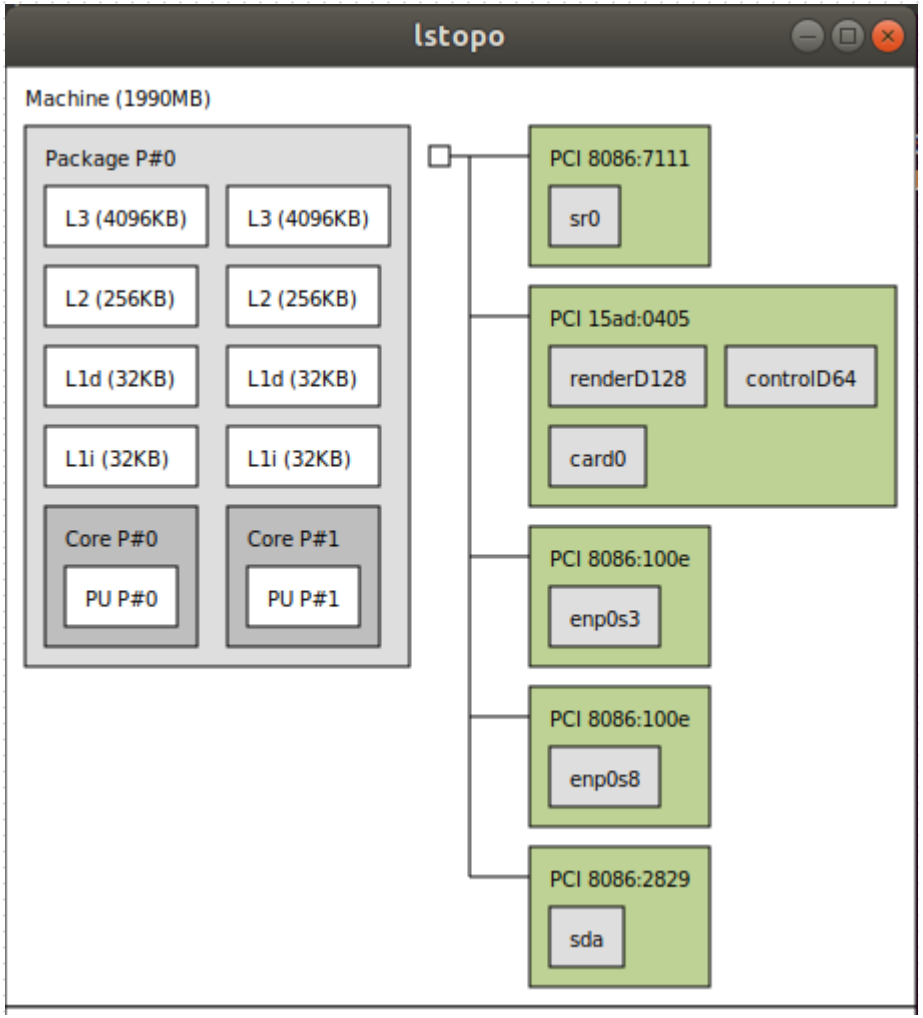
Memory hierarchy of an AMD Bulldozer server : microarchitecture

Source: https://upload.wikimedia.org/wikipedia/commons/9/95/Hwloc.png



Linux Kernel : Memory Management

- **Can see any system's CPU/cache 'topology' in a fantastic graphical manner**
- **How? Install and run the *lstopo(1)* utility**
  - Ubuntu: sudo apt install hwloc
  - (also use the CLI hwloc-* utils!)
  - lstopo



Linux Kernel : Memory Management

- **Lookup CPU cache-line size**

On x86_64:
$ getconf -a | grep "CACHE_LINESIZE"

LEVEL1_ICACHE_LINESIZE 64

LEVEL1_DCACHE_LINESIZE 64

LEVEL2_CACHE_LINESIZE 64

LEVEL3_CACHE_LINESIZE 64

LEVEL4_CACHE_LINESIZE

- ***On ARM-32, via the DT! (as an eg. for a Broadcom SoC here)***

```
dts $ grep -Hn -B8 "cache-line" bcm*.dts*
bcm63138.dtsi-77-
bcm63138.dtsi-78-               L2: cache-controller@1d000 {
bcm63138.dtsi-79-                       compatible = "arm,pl310-cache";
bcm63138.dtsi-80-                       reg = <0x1d000 0x1000>;
bcm63138.dtsi-81-                       cache-unified;
bcm63138.dtsi-82-                       cache-level = <2>;
bcm63138.dtsi-83-                       cache-size = <524288>;
bcm63138.dtsi-84-                       cache-sets = <1024>;
bcm63138.dtsi:85:                       cache-line-size = <32>;
dts $
```

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)

[see next slide for an example]

- an **LLC (Last Level Cache) Miss** is expensive!
- use profilers (perf and **eBPF** tools (*bcc*)) 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

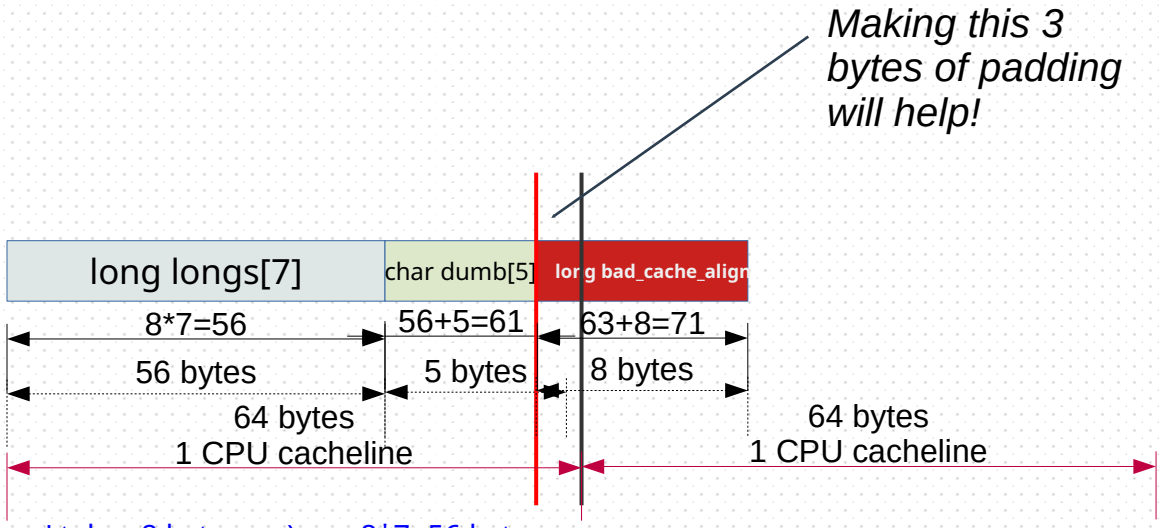
>>

Linux Kernel : Memory Management

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)

```
$ getconf -a|grep CACHE_LINESIZE
LEVEL1_ICACHE_LINESIZE 64
LEVEL1_DCACHE_LINESIZE 64
LEVEL2_CACHE_LINESIZE 64
LEVEL3_CACHE_LINESIZE 64
LEVEL4_CACHE_LINESIZE 0
$
```




```
struct faker {
        long longs[7]; // (on x86_64 a 'long' takes 8 bytes, so) use 8*7=56 bytes
        char dumb[5]; // use 56+5=61 bytes; now 3 bytes left in this cacheline!
        long bad_cache_align; /* use 61+8=69 bytes, thus spilling over the
                cacheline! Poor for performance – 2 accesses required to read this member ! */
};
```

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

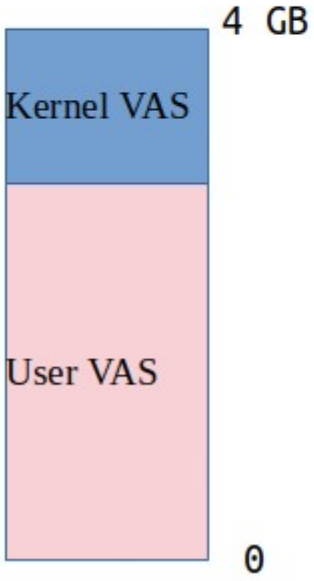
  On a 32-bit ARM Linux system
  2 GB : 2 GB :: user-space : kernel-space
  [sometimes 3 GB : 1 GB]

- *So, use a 32 or 64 bit processor for running Linux?*
  - Embedded: <= 1 GB RAM, 32-bit is ok (though limitations still crop up)
  - > 1 GB RAM: do switch to 64-bit

Linux Kernel : Memory Management

VM split is a kernel configurable !
In ./arch/x86/Kconfig: VMSPLIT_3G / VMSPLIT_2G / VMSPLIT_1G

x86_32 : VM split is at 3 GB
ARM-32 : VM split is at either 2 GB or 3 GB



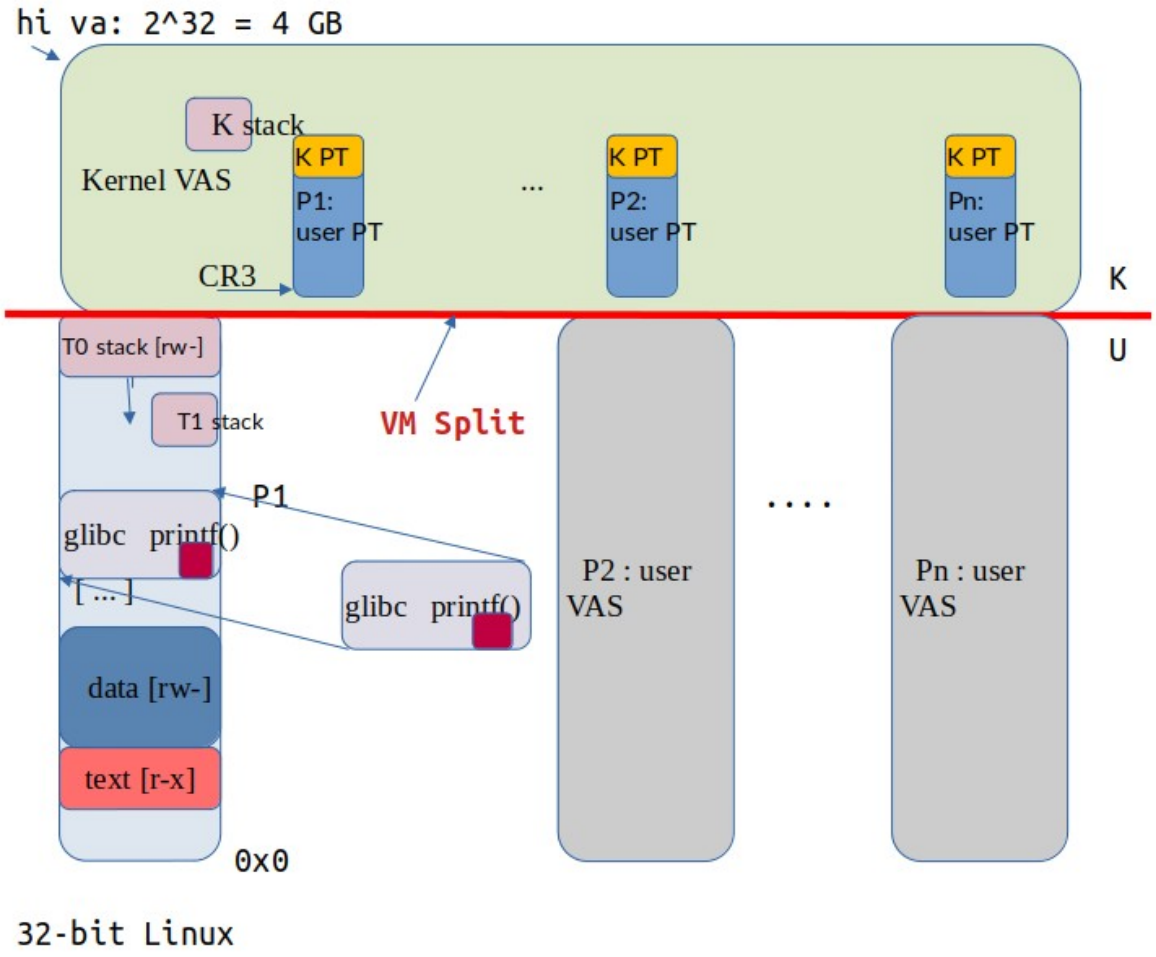
Linux Kernel : Memory Management

How the VM Split works…

On 32-bit Linux, as a simpler example to begin with…

Legend:

K = kernel
PT = page tables



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}

Linux Kernel : Memory Management

Common powers-of-2 (good to know!)

- **2^10 = 1024 (~= 10^3)**
- **2^20 = 2^10 * 2^10 = 1024*1024 = 1,048,576 = 1 MB (~= 1 million)**

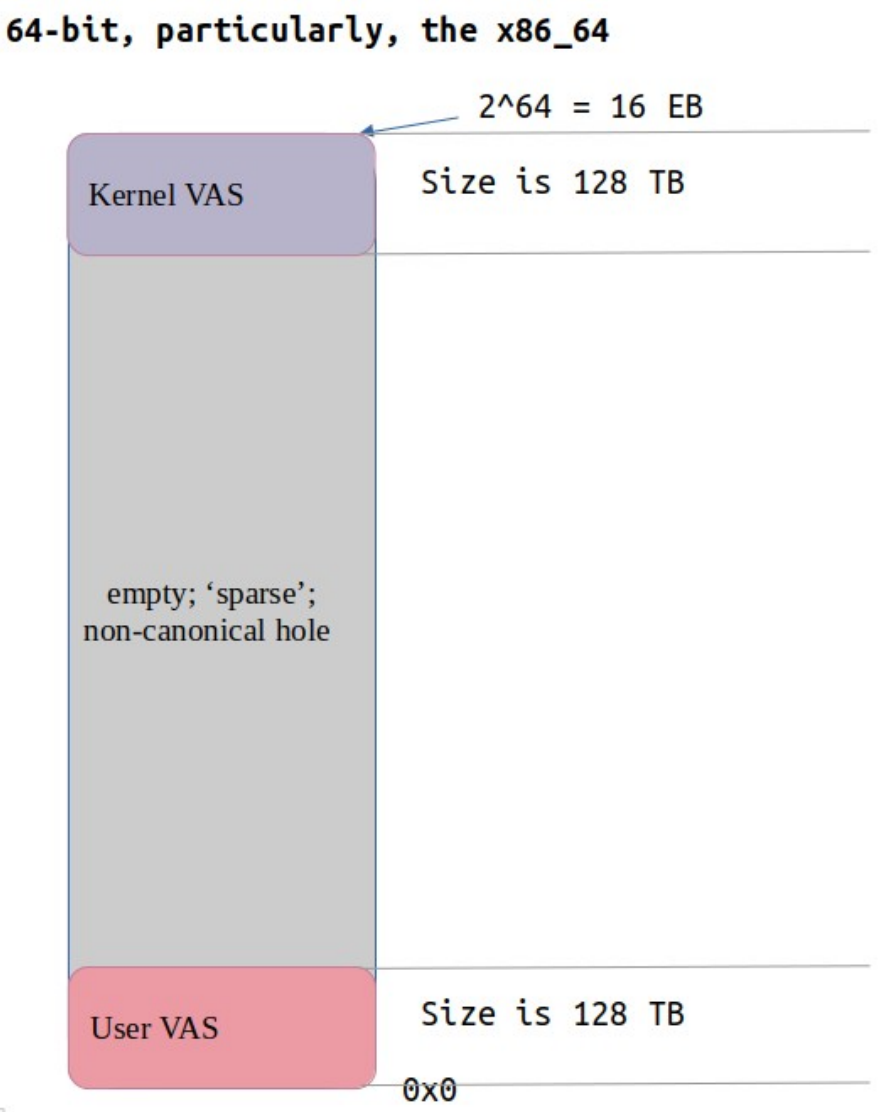
- **2^32**
  - Why 2^32? As this is the size of the total VAS per process on a 32-bit system !
  - it's 2^10*2^10*2^12 = 1024*1024*4096 = 4,294,967,296 = **4 GB**
- **2^64**
  - Why 2^64? As this is the size of the total VAS per process on a 64-bit system !
  - 2^64 = 16 exabytes (EB) = 16,384 PB = 16,777,216 TB = 17,179,869,184 GB !

    (= ~16.7 million TB = ~17.2 billion GB = 18,446,744,073,709,552,000).

Linux Kernel : Memory Management

64-bit Linux

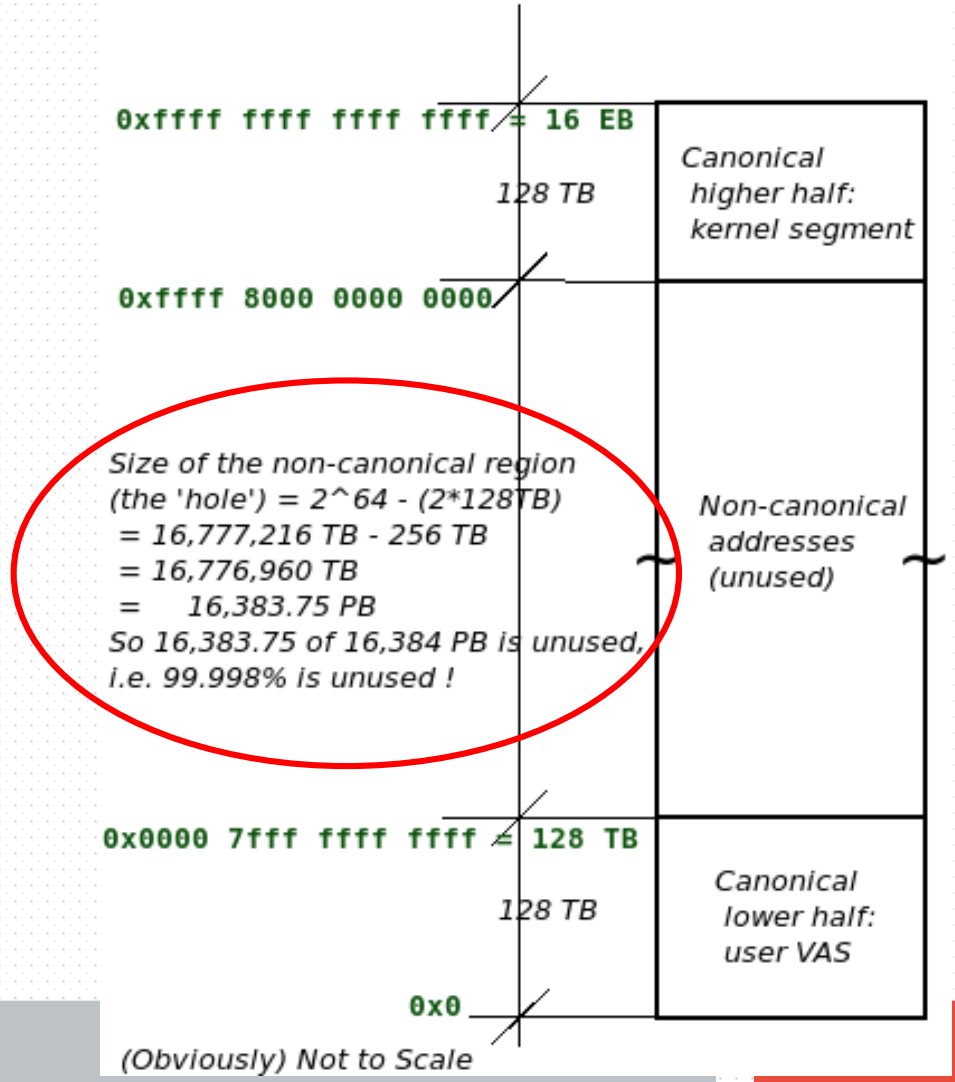
AMD64/x86_64
full process VAS



Linux Kernel : Memory Management

So, the 64-bit VAS on the x86_64 – with typically 48-bit addressing – is as follows:

(Diagram from the LKP-2E book)



64-bit VAS – can be done in several ways

As of now, on x86_64, 48-bit addressing is the standard (default)

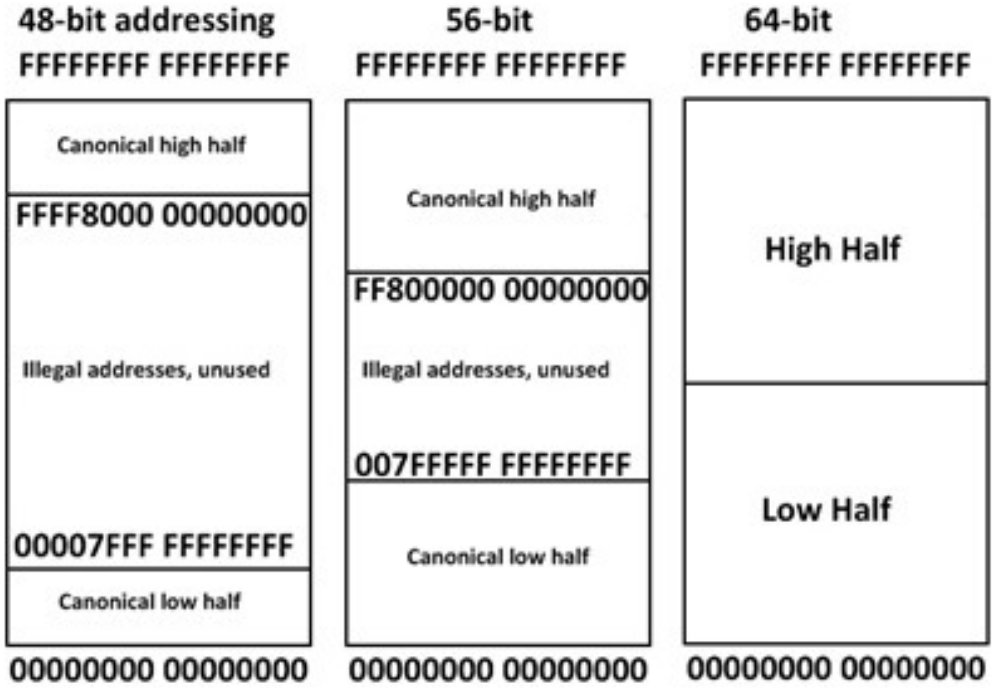


Figure 2 - *Memory Addressing*

Pic: Professional Linux Kernel Architecture, Mauerer

Linux Kernel : Memory Management

PAGE_OFFSET

Splitting point, or, on 64-bit, the start of the kernel segment, is called PAGE_OFFSET

- More technically, it's the 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
$
```

Linux Kernel : Memory Management

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

Arch-specific VM Splits (from the LKP 2E book)

Row #	Arch	N-Level	# Addr Bits	Total in-use VAS (2^ addr-bits)	VM Split U : K	User-space		Kernel-space
						Start UVA	End UVA	Start KVA (with End KVA always = 0xffff ffff ffff ffff)
1	IA-32	2	32	4 GB	3 GB : 1 GB	0x0	0xbfff ffff	0xc000 0000
2	ARM (AArch32)	2	32	4 GB	2 GB : 2 GB (or 3GB : 1GB)	0x0	0x7fff ffff	0x8000 0000
3	x86_64	4	48	256 TB	128 TB : 128TB	0x0	0x0000 7fff ffff ffff	0xffff 8000 0000 0000
4		5	57	128 PB	64 PB : 64 PB	0x0	0x00ff ffff ffff ffff	0xff00 0000 0000 0000
5	AArch64	3	40	1 TB	512 GB : 512GB	0x0	0x0000 7fff ffff ffff	0xffff ff80 0000 0000
6		4	49	512 TB	256 TB : 256TB	0x0	0x0000 ffff ffff ffff	0xffff 0000 0000 0000
7	ARMv8.2 LPA	3	53	8 PB	4 PB : 4 PB	0x0	0x0010 0000 0000 0000	0xfff0 0000 0000 0000

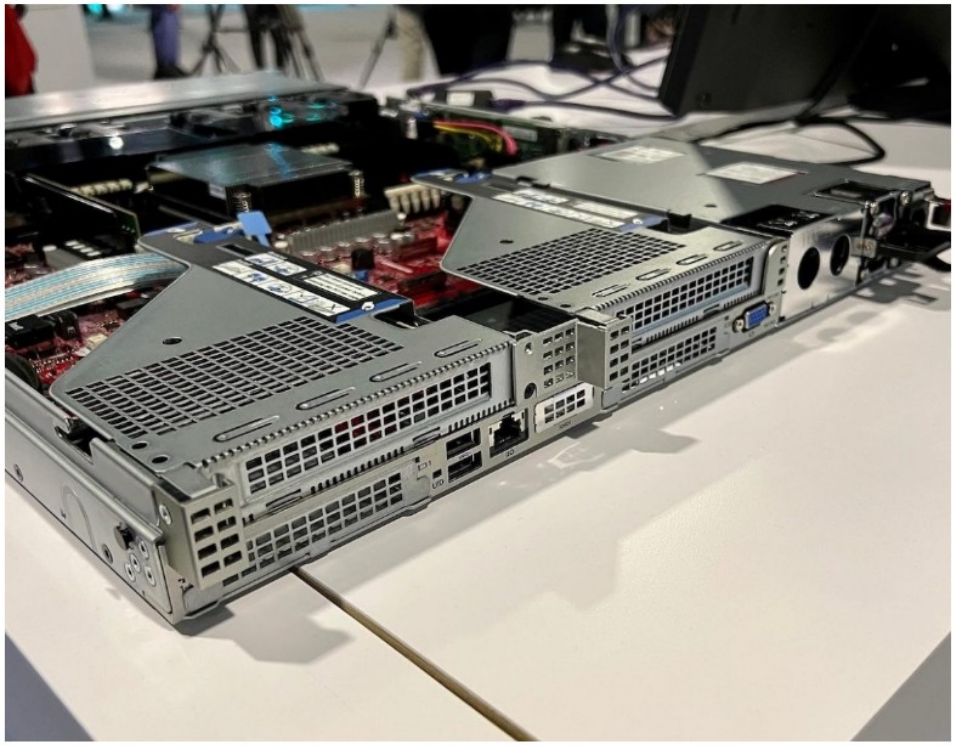
- Row #1 and row #2: the 32-bit IA-32 and AArch32 with a 3:1 and 2:2 (HB) VM split respectively
- Row #3: the common case: the x86_64 with 4-level paging and 48 (LSB) bits used for addressing
- Row #4: 5-level paging is possible but requires 4.14 Linux or later
- Row #5: AArch64 example: with a standard Yocto 4.0.4 (Kirkstone) build for Poky distro, machine raspberrypi4-64 (with default config value `CONFIG_ARM64_VA_BITS=39`; thus 40 address bits configured, total VAS is 2^40 = 1 TB, thus, here, both the kernel and user VAS is 512 GB each)
- Row #6: AArch64: above + config `CONFIG_ARM64_VA_BITS=48`; thus 49 (0 to 48) address bits configured, total VAS is 2^49 = 512 TB, thus, here, both the kernel and user VAS is 256 TB each
- Row #7: This config requires (AArch64) ARMv8.2 LPA (Large Physical Addresses) extension + >= 5.4 Linux + 64K page size (for 3-level paging) enabled. It results in 53 address bits being configured, total VAS is 2^53 = 8 PB, thus, here, both the kernel and user VAS is 4 PB each.

ARM-64 (AArch64) is indeed being used for servers

An example (Nov 2022):

HPE ProLiant RL300 Gen11 Data sheet (*link*): use cases are datacenter and cloud.

(Max memory: ‘4.0 TB with 256 GB DDR4’).



Related: ***HPE ProLiant RL300 Gen11 server certified with Oracle Linux to help customers deliver cloud native solutions, Resta, July 2023 (Oracle Linux blog)***
[link]

Linux Kernel : Memory Management

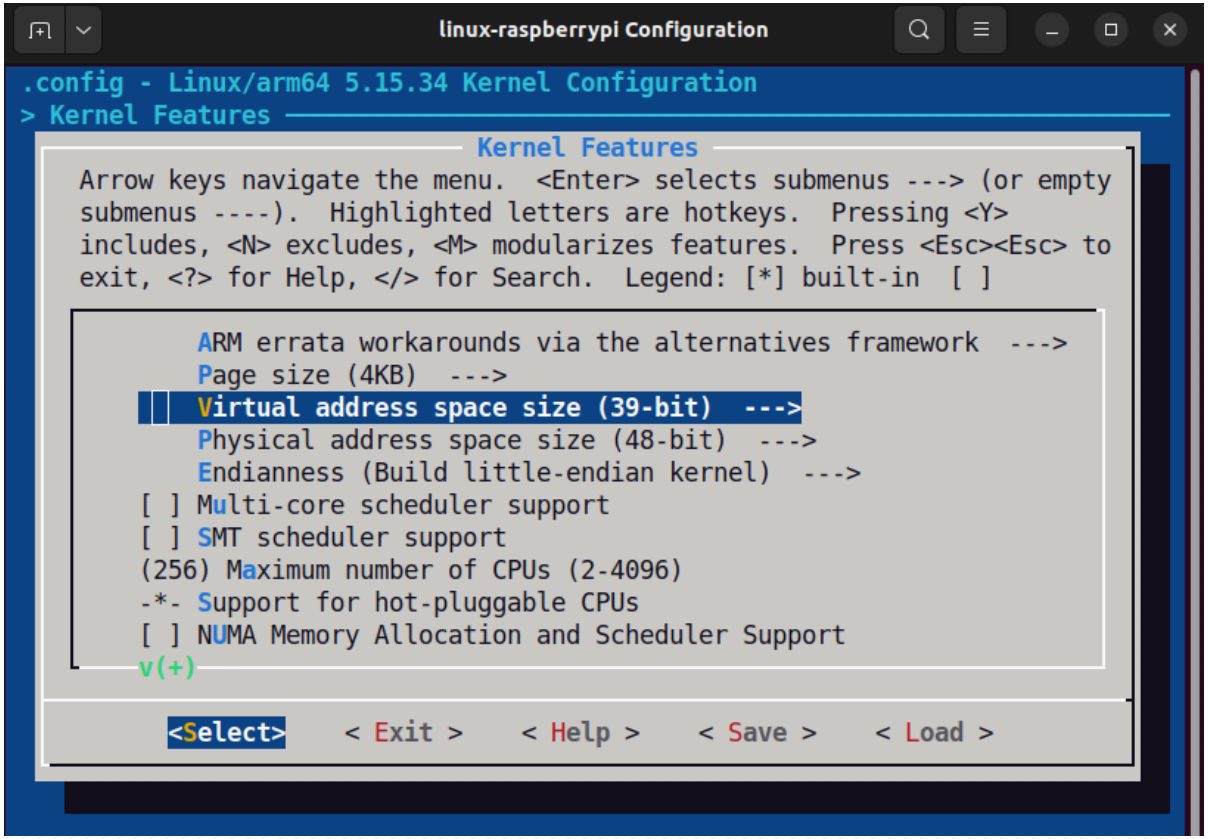
Arch-specific VM Splits – Customizing them

How **can one change the usable VAS (and possibly physical) sizes on 64-bit?**

Taking the Aarch64 (particularly for the Raspberry Pi BCM2837 SoC) as an example:

Menu options:

- **Kernel Features | Virtual address space size (39-bit)**
  - **CONFIG_ARM64_VA_BITS**
  - **(other choice here is 48-bit)**
- **Physical address space size (48-bit)**



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, it does *not* hoard it!
- Gives it to userspace on demand (via demand paging)

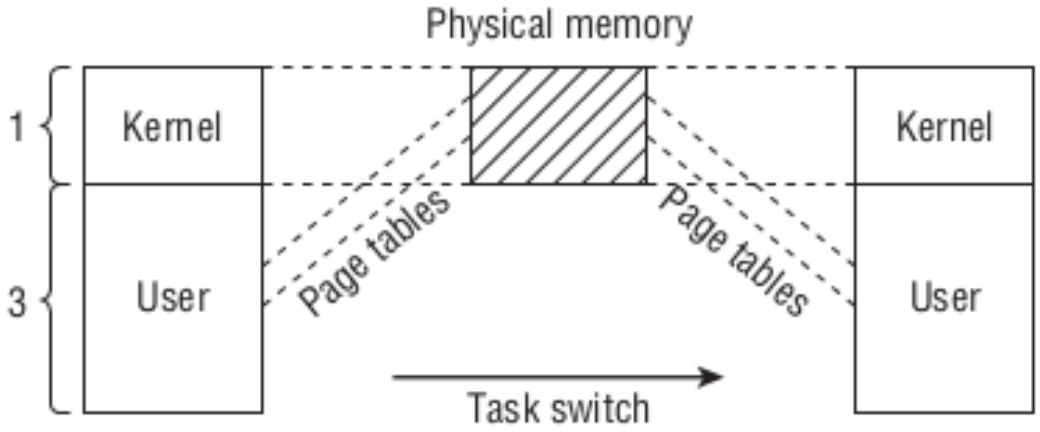


Figure 3-14: Connection between virtual and physical address space on IA-32 processors.

Pic: Professional Linux Kernel Architecture, Mauerer

Linux Kernel : Memory Management

- Physical – platform – RAM is “direct-mapped” into the kernel segment at boot

- So, the max amount of RAM would be thus limited by the virtual space available in the kernel VAS!
  - Indeed: it’s called the ‘**lowmem**’ region

  - **On 32-bit**, it can (perhaps easily) exceed 768 MB causing the need for the zone named ZONE_HIGHMEM
  - Mitigated by using the PAE (Physical Address Extension) on 32-bit; can then use upto 64 GB RAM

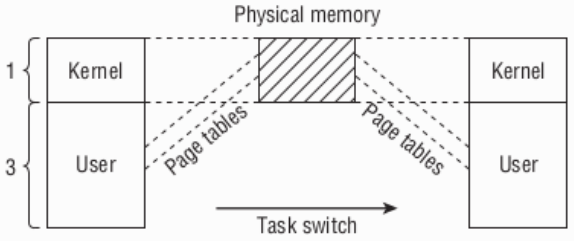


Figure 3-14: Connection between virtual and physical address space on IA-32 processors.

Linux Kernel : Memory Management

- Physical – platform – RAM is “direct-mapped” into the kernel segment at boot

- On 64-bit, this is usually no problem, as the kernel VAS is routinely very large (anywhere from 256 GB to 64 PB! (see the *Arch-specific VM Splits* slide))

- Q. How much RAM max is actually used on a single node?
  A. See *Pleiades Supercomputer (NASA)*
  - the max RAM used here seems to be 128 GB per node!
  - x86_64 Linux has 128 TB (131,072 GB) of user and kernel VAS
  - Try (on x86_64):
    sudo dmidecode -t 16 |grep "Maximum Capacity"
    Maximum Capacity: 128 GB

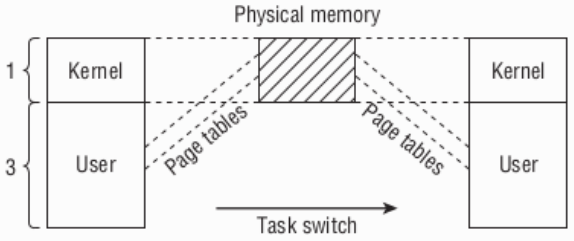


Figure 3-14: Connection between virtual and physical address space on IA-32 processors.

FYI / Optional

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**

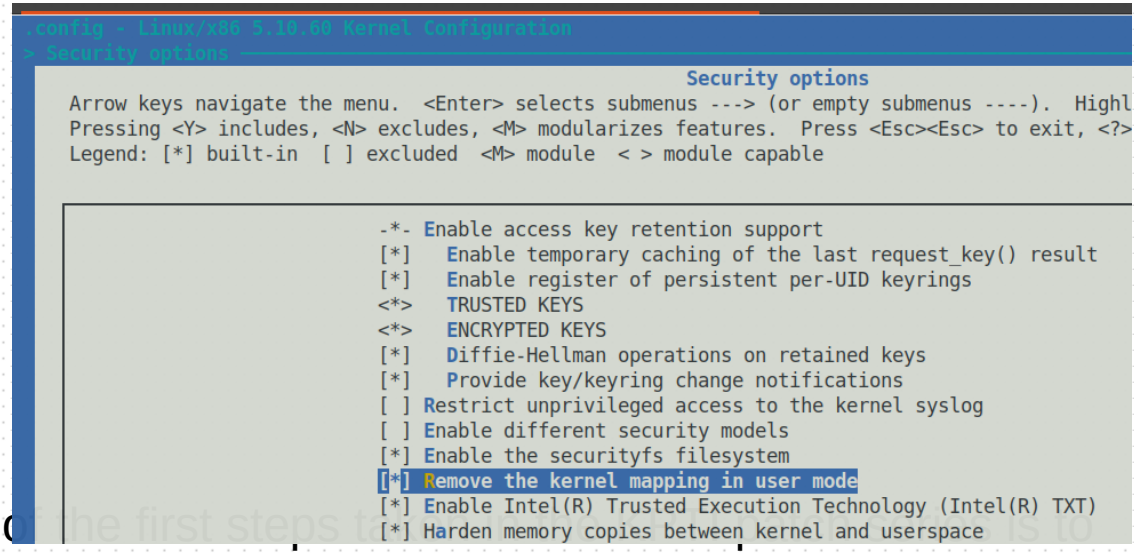
```
$ uname -a
Linux ... 4.14.11-300.fc27.x86_64 #1 SMP Wed Jan 3 13:52:28 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
$ dmesg |grep "isolation"
Kernel/User page tables isolation: enabled
$ dmesg |grep -i spectre
kern  :info  : [Sat Mar  3 15:16:00 2018] Spectre V2 : Mitigation: Full generic retpoline
kern  :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 **FYI / Optional**

- 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 (CONFIG_PAGE_TABLE_ISOLATION)!
- **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 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 **FYI / Optional**

- Mitigating the detrimental slowdowns due to the page table isolation (KPTI) solution
- From Brendan Gregg's presentation 'YOW! 2018 Brendan Gregg - Cloud Performance Root Cause Analysis at Netflix' (*link*; ~ 56:40)

PMCs: EC2 Nitro Hypervisor

- Some instance types (large, Nitro-based) support most PMCs!
- Meltdown KPTI patch TLB miss analysis on a c5.9xl:

nopti:

```
# tlbstat -C0 1
K_CYCLES    K_INSTR      IPC DTLB_WALKS ITLB_WALKS K_DTLBCYC  K_ITLBCYC  DTLB% ITLB%
2854768     2455917     0.86 565        2777       50         40          0.00  0.00
2884618     2478929     0.86 950        2756       6          38          0.00  0.00
2847354     2455187     0.86 396        297403     46         40          0.00  0.00
[...]
```

pti, nopcid:

```
# tlbstat -C0 1
K_CYCLES    K_INSTR      IPC DTLB_WALKS ITLB_WALKS K_DTLBCYC  K_ITLBCYC  DTLB% ITLB%
2875793     276051      0.10 89709496   65862302   787913     650834     27.40 22.63
2860557     273767      0.10 88829158   65213248   780301     644292     27.28 22.52
2885138     276533      0.10 89683045   65813992   787391     650494     27.29 22.55
2532843     243104      0.10 79055465   58023221   693910     573168     27.40 22.63  worst case
[...]
```

Linux Kernel : Memory Management

Meltdown/Spectre Effects Mitigation **FYI / Optional**

- 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

- **Use lscpu to see if your CPU(s) are vulnerable !**
- [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

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!)
- x86_64 supports 4K, 2M and even 1GB size pages
- Kernel support: CONFIG_HUGETLBFS
- Stats: grep HugePages /proc/meminfo
- 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 showing kernel VM layout

Page Cache -diagram, blog article [mmap]

OOM killer

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

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

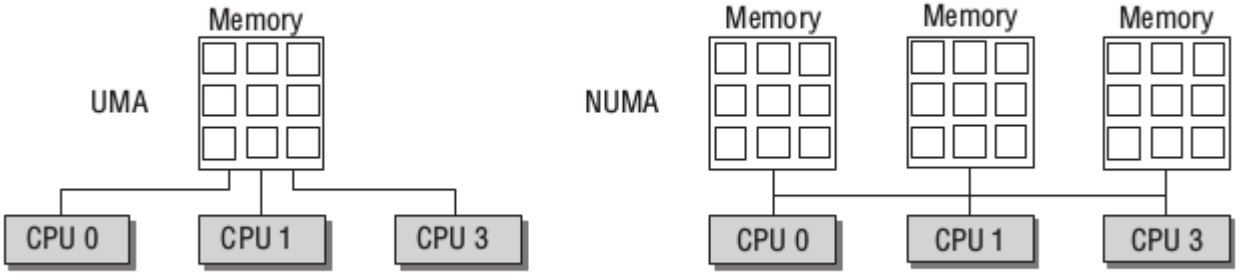


Figure 3-1: UMA and NUMA systems.

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

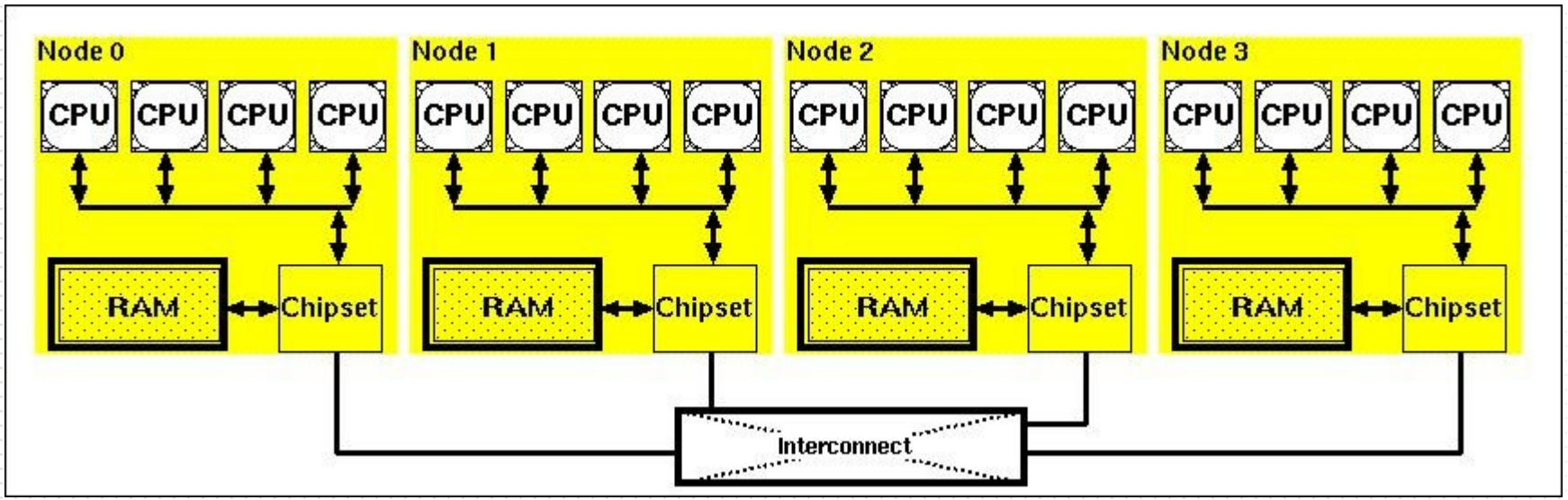
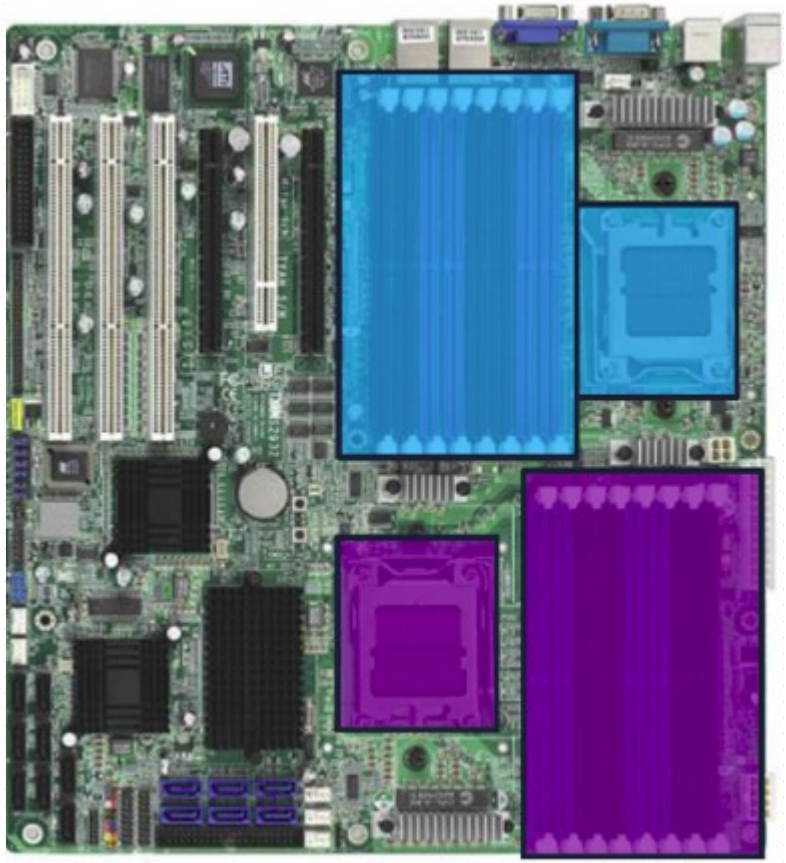


Image Source

Linux Kernel : Memory Management

NUMA server



Linux Kernel : Memory Management

Actual NUMA servers

NUMA-Q, IBM

Integrity SuperdomeX, HPE

Linux Kernel : Memory Management

Linux OS; at boot:

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*

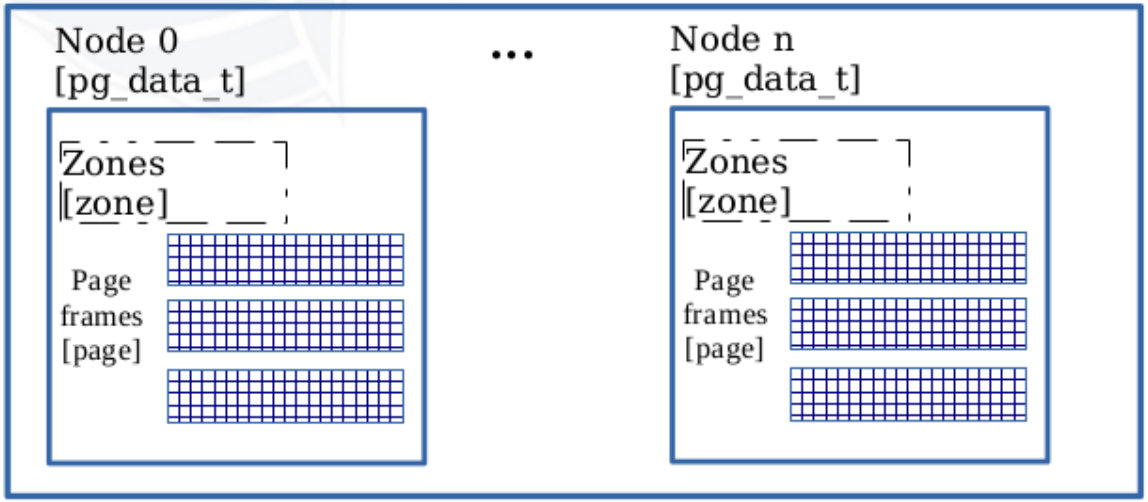
Each node is further split into (at least one, upto four) *zones*

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
```

Linux Kernel : Memory Management

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 when threads on different processors modify variables that reside on the same cache line. 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

Linux Kernel : Memory Management

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

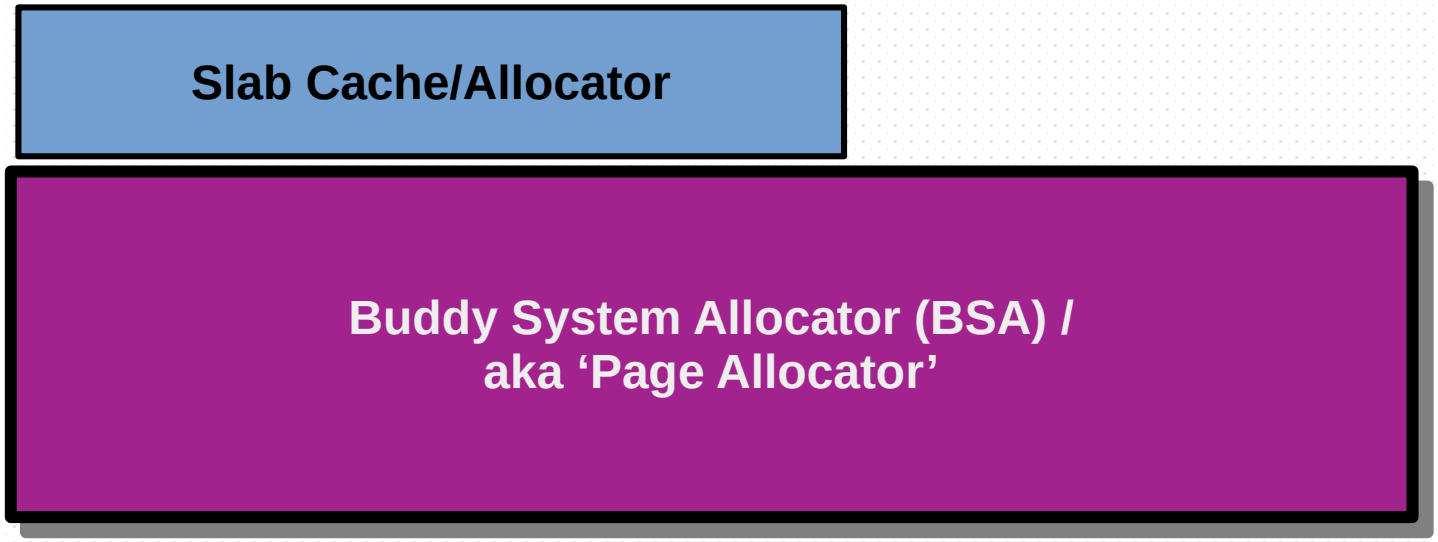
```
struct zone {
[…]
        spinlock_t              lock;
--snip--
         struct free_area        free_area[MAX_ORDER];
--snip--
       ZONE_PADDING(_pad1_)

        /* Fields commonly accessed by the page reclaim scanner */
         spinlock_t              lru_lock;
<<
 File : [3.10.24] : include/linux/mmzone.h
  90 /*
  91  * zone->lock and zone->lru_lock are two of the hottest locks in the kernel.
  92  * So add a wild amount of padding here to ensure that they fall into separate
  93  * cachelines.  There are very few zone structures in the machine, so space
  94  * consumption is not a concern here.
  95  */
>>
...
```

Linux Kernel : Memory Management

Linux Kernel Memory Allocation

- Layered Approach



Linux Kernel : Memory Management

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 set to 11); thus 11 elements in the array
- 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

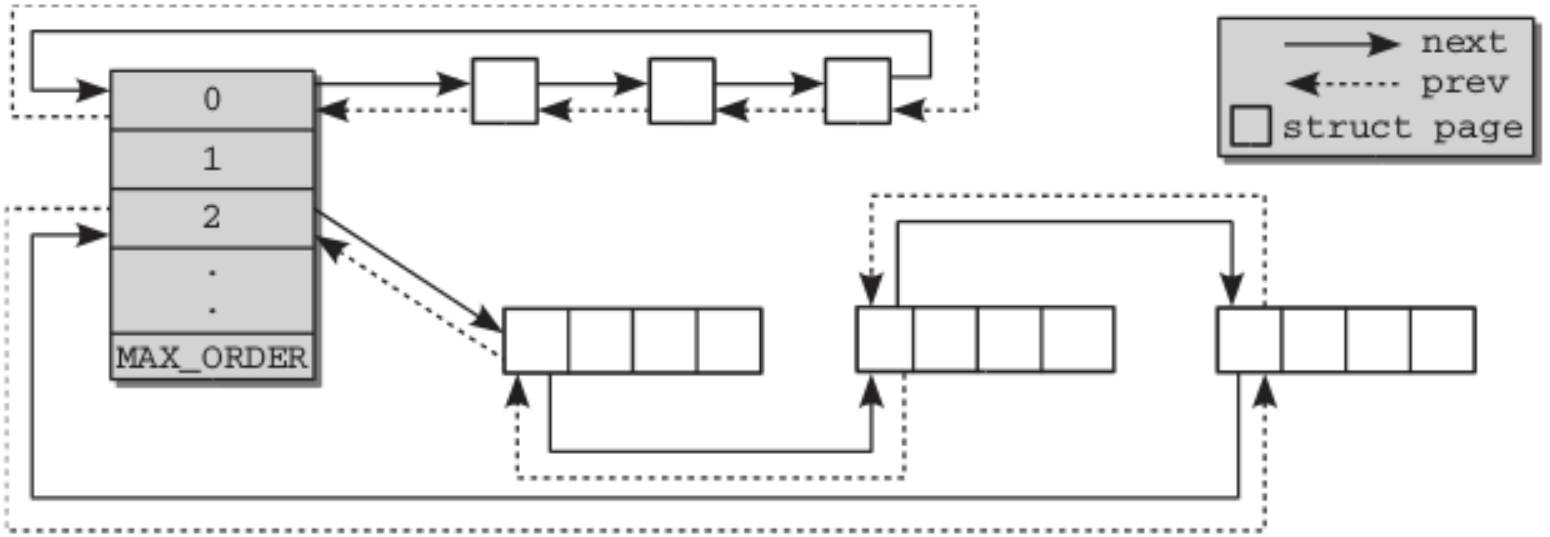


Figure 3-22: Linking blocks in the buddy system.

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:

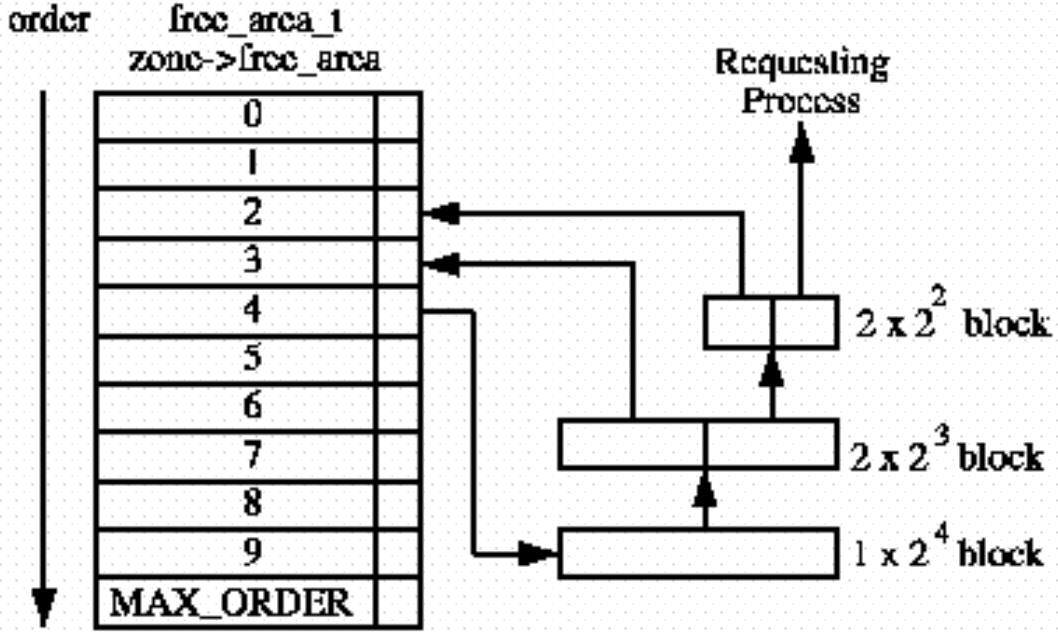
```
$ cat /proc/buddyinfo
Node 0, zone      DMA      2      3      6      2      3      1      0      0      1      1      3
Node 0, zone    DMA32  14617   3480   1303    465    388     97     63     23     11      3      0
Node 0, zone   Normal   1208   1909    303    609    552    180     49     10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

```
Order:              0      1      2      3      4      5      6      7      8      9     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* freelist ach node)!

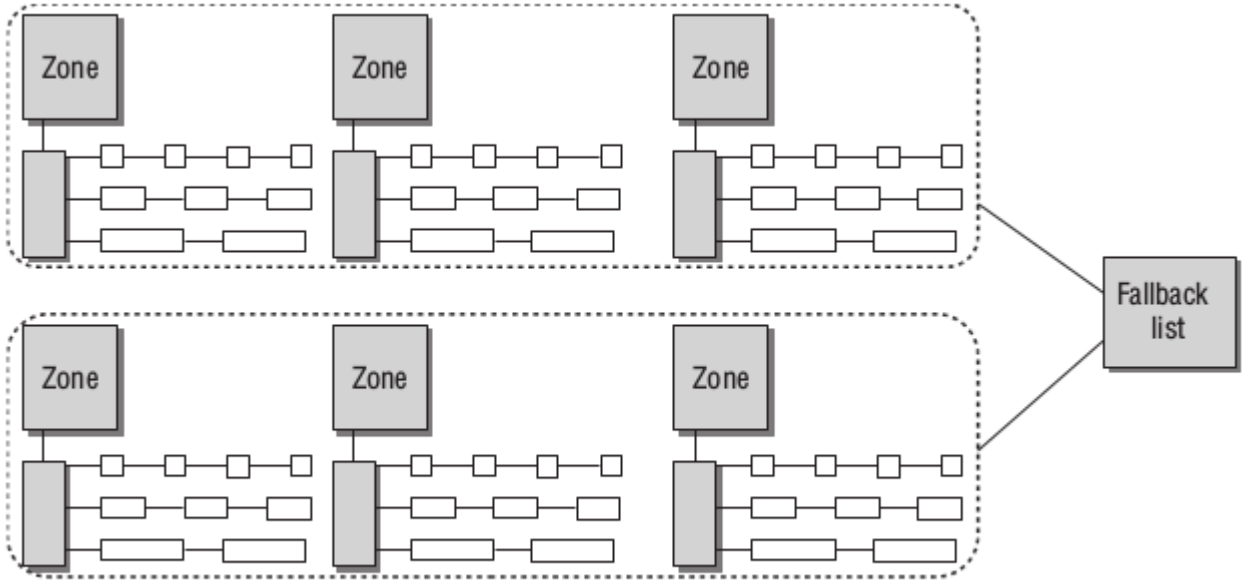


Figure 3-23: Relationship between buddy system and memory zones/nodes.

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

[- mitigated by using the alloc_pages_exact()]

Linux Kernel : Memory Management

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

FYI, slab (SLUB) Internals: 'Linux SLUB Allocator Internals and Debugging, Part 1 of 4, Imran Khan, Dec 2022' [*link*]

Linux Kernel : Memory Management

Check out the available slabs for the kmalloc:

```
                          Num   Total  Size   Pages
$ sudo vmstat -m | grep "^kmalloc-[0-9].."
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 (i.e. # of objects currently allocated from this cache)
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);
```

Kernel Dynamic Memory Alloc APIs

BSA - 'low-level' APIs

```
void *alloc_pages_exact(size_t size, gfp_t gfp_mask);
void free_pages_exact(void *virt, size_t size);
```

The **`alloc_pages_exact(size, gfp_mask)`** optimizes memory like this:

- It first allows the BSA to allocate memory in the usual manner (by invoking `__get_free_pages(…)`)
- It then seeks to the amount required (size), and then goes in a loop freeing pages as much as is possible.

Eg. `alloc_pages_exact(68*1024, GFP_KERNEL);`

Requests 68 KB but, initially, the unoptimized BSA allocates 128 KB. So, it seeks to the 68 KB position and starts freeing up memory pages from there!

```
^                    ^<... freed...  ... >^
0                    68K                 128K
```

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 (spinlock critical section, 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!!
(Further, even the alloc_pages_exact() API's granularity is 1 page).

```
#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 kfree_sensitive(const void *p);   /* zero the mem via memzero_explicit()
        and then free it (security); renamed to kfree_sensitive() from kzfree() in 5.9 */
void kzfree(const void *objp);
```

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)!

Linux Kernel : Memory Management

Runtime Usage of kmalloc

How many bytes are typically allocated by callers to kmalloc() ?

- Older way – use Jprobes (jumper probes); deprecated in 4.15
- Modern way – eBPF tooling ! (Debian/Ubn: sudo apt install bpfcc-tools)

Use argdist[-bpfcc]

```
$ sudo argdist-bpfcc -H 'p::__kmalloc(u64 size):u64:size' 2>/dev/null
[...]
[07:26:31]
     size               : count     distribution
        0 -> 1          : 0         |                                        |
        2 -> 3          : 0         |                                        |
        4 -> 7          : 194       |**********                              |
        8 -> 15         : 102       |*****                                   |
       16 -> 31         : 183       |*********                               |
       32 -> 63         : 74        |****                                    |
       64 -> 127        : 737       |****************************************|
      128 -> 255        : 24        |*                                       |
      256 -> 511        : 0         |                                        |
      512 -> 1023       : 35        |*                                       |
     1024 -> 2047       : 1         |                                        |
[...]
$
```

Interesting! The bucket used most frequently is 64 to 127 bytes !

Linux Kernel : Memory Management

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 11 (0-10), the upper limit is 4 MB

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

Linux Kernel : Memory Management

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)
```

Thus, ksize() helps us detect the wastage (internal fragmentation)!

Linux Kernel : Memory Management

Demo: the https://github.com/kaiwan/L2_kernel_trg/tree/master/mm/allocsize_tests/ksize_test kernel module.

*This serves to demo the wastage issue that both the page allocator (and hence the slab allocator) suffer from – **internal fragmentation or wastage!***

Alternate way to check which slabs suffer from wastage: use the kernel's ***slabinfo*** utility:

$ sudo slabinfo –help
…
-L|--Loss Sort by loss
…

Tip-
Lookup the discussion in my *LKD* book, *Ch 6 – Debugging Kernel Memory Issues, Part 2* section *'Learning how to use the slabinfo and related utilities'*!

```
~ $ sudo slabinfo -L|head
Name                   Objects Objsize          Loss Slabs/Part/Cpu  O/S O %Fr %Ef Flg
ext4_inode_cache        336433    1168         25.2M   12733/748/29   27 3   5  93 a
dentry                  247059     192         13.1M 14676/6386/108   21 0  43  78 a
radix_tree_node         906927     576          9.3M   32364/484/91   28 2   1  98 a
nvidia_stack_cache         737   12288          3.0M       348/2/22    2 3   0  74
ext4_extent_status       37560      40          2.1M    771/661/122  102 0  74  41 a
kmalloc-192              20809     192          1.7M   1246/875/148   21 0  62  69
kmalloc-256              53420     256          1.6M    1793/420/82   32 1  22  89
:a-0000104             1087788     104          1.1M    27669/0/223   39 0   0  99 *a
vm_area_struct          159040     208        629.7K   3956/160/159   39 1   3  98
~ $
```

Linux Kernel : Memory Management

Recent: helping identify kmalloc() wastage ; from 6.1

Commit ID: 6edf2576a6cc

Ref:
https://github.com/torvalds/linux/commit/6edf2576a6cc46460c164831517a36064eb8109c

Documentation/mm/slub.rst

…

DebugFS files for SLUB
==========

For more information about current state of SLUB caches with the user tracking debug option enabled, debugfs files are available, typically under **/sys/kernel/debug/slab/<cache>/** (created only for caches **with enabled user tracking**). There are 2 types of these files with the following debug information:

1. **alloc_traces**::

   Prints information about unique allocation traces of the currently allocated objects. The output is sorted by frequency of each trace.

[...]

Linux Kernel : Memory Management

Recent: helping identify kmalloc() wastage ; from 6.1

Commit ID: 6edf2576a6cc

Ref:
https://github.com/torvalds/linux/commit/6edf2576a6cc46460c164831517a36064eb8109c

...
We've met a kernel boot OOM panic (v5.10), and from the dumped slab info:

[26.062145] kmalloc-2k 814056KB 814056KB

From debug we found there are huge number of 'struct iova_magazine', whose size is 1032 bytes (1024 + 8), so each allocation will waste 1016 bytes. Though the issue was solved by giving the right (bigger) size of RAM, it is still nice to optimize the size (either use a kmalloc friendly size or create a dedicated slab for it).

And from lkml archive, there was another crash kernel OOM case [1]
...

Slab allocator

- From what we just learned, keeping structure sizes the same as generic slab cache sizes would be the ideal case
- This is often done, by design…
- Can check via the useful 'crash' tool:

Using crash's *whatis* command, show all structures that have a size of exactly 64 bytes:

```
crash> whatis -r 64
SIZE  TYPE
  64  _kvm_stats_desc
  64  acct_v3
  64  acpi_gpio_event
  64  acpi_gpio_lookup
[ ... ]
```

(With a 6.1.25 kernel, I got a total of 300 structures!).

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 required, unlike the kmalloc)

- the page frames corresponding to the virtual memory region allotted are indeed allocated immediately (under the hood, via alloc_pages() - page allocator!)

Linux Kernel : Memory Management

vmalloc

- APIs

```
void *vmalloc(unsigned long size);
void *vzalloc(unsigned long size);  // recommended !

void vfree(const void *addr);
```

vmalloc: example code

fs/cramfs/uncompress.c

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

<< see next slide >>

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);
void kvfree(const void *addr);
```

Linux Kernel : Memory Management

The vmalloc() is used by the kernel to allocate thread stacks when VMAP_STACK=y.

This is advantageous for security…

```
config VMAP_STACK
    default y
    bool "Use a virtually-mapped stack"
    depends on HAVE_ARCH_VMAP_STACK && !KASAN
    ---help---
      Enable this if you want the use virtually-mapped kernel stacks
      with guard pages.  This causes kernel stack overflows to be
      caught immediately rather than causing difficult-to-diagnose
      corruption.

      This is presently incompatible with KASAN ...
```

Linux Kernel : Memory Management

[OLDER / deprecated] 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)
```

-the 'prot' param was removed in 5.8, as only valid prot now is PAGE_KERNEL

Linux Kernel : Memory Management

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 :

https://github.com/PacktPublishing/Linux-Kernel-Programming/tree/master/ch9/slab_custom >>

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 (1)*

- Devres : *device resources* manager
- A framework developed for the device model, enabling auto-freeing of memory allocated via the 'devres' APIs
- The freeing occurs on driver detach
- *Requires* the struct device pointer though

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

Managed memory for drivers with the devres APIs (2)

- DON'T try and replace all allocations with the 'managed' variant APIs
- They are best suited to device driver *probe()* methods, ensuring correct initialization, and if it fails, auto-freeing of resources ensuring correct de-init
- *Ref:*
  "… Please note that managed resources (whether it's memory or some other resource) are meant to be used in code responsible for the probing the device. They are generally a wrong choice for the code used for opening the device, as the device can be closed without being disconnected from the system. Closing the device requires freeing the resources manually, which defeats the purpose of managed resources.

  The memory allocated with kzalloc() should be freed with kfree(). The memory allocated with devm_kzalloc() is freed automatically. It can be freed with devm_kfree(), but it's usually a sign that the managed memory allocation is not a good fit for the task. …"

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

Linux Kernel : Memory Management

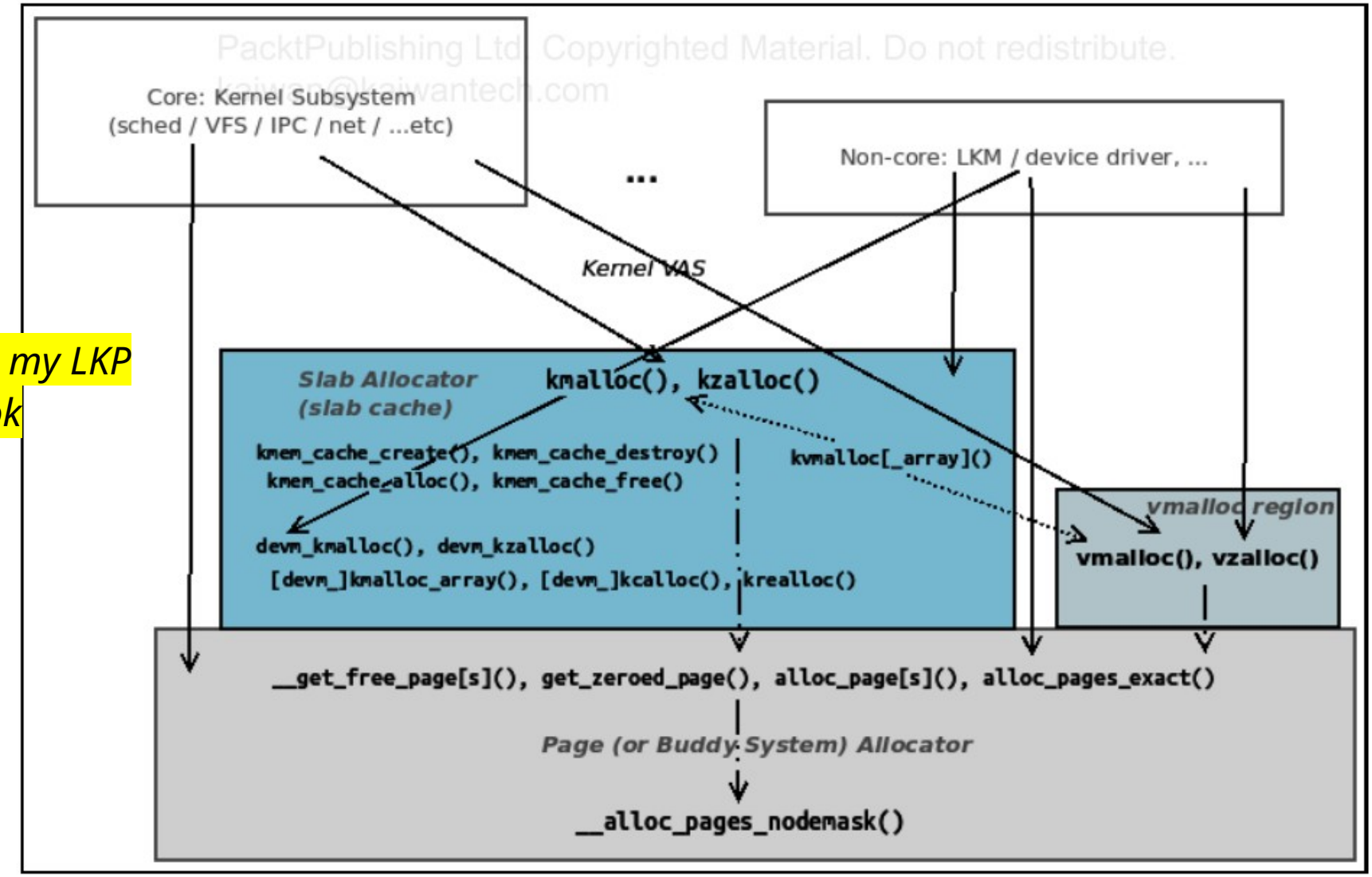
Dynamic Memory Allocation APIs to Use

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` *`(kzalloc() preferred)`*
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

Dynamic Memory Allocation APIs to Use

The following is a diagram showing several of the (exposed to module / driver authors) kernel memory allocation APIs:



Src: my LKP book

Figure 9.6 – Conceptual diagram showing the kernel's memory allocation API set (for module / driver authors)

Linux Kernel : Memory Management

Dynamic Memory Allocation APIs to Use

Track kernel-level allocations (that aren't freed yet – a way to look for leaks!), and see the call stack – using the memleak eBPF tool (roughly every 5th allocation)!

```
 $ sudo memleak-bpfcc -s 5 2>/dev/null
Attaching to kernel allocators, Ctrl+C to quit.
[16:26:32] Top 10 stacks with outstanding allocations:
384 bytes in 6 allocations from stack
kmem_cache_alloc_node+0x35e [kernel]
kmem_cache_alloc_node+0x35e [kernel]
alloc_vmap_area+0x3ee [kernel]
__get_vm_area_node.constprop.0+0xba [kernel]
__vmalloc_node_range+0x59 [kernel]
dup_task_struct+0x248 [kernel]
copy_process+0x1e2 [kernel]
kernel_clone+0x9d [kernel]
__do_sys_clone+0x5d [kernel]
__x64_sys_clone+0x25 [kernel]
do_syscall_64+0x61 [kernel]
entry_SYSCALL_64_after_hwframe+0x44 [kernel]
```

Tip:
memleak[-bpfcc] -h
and see https://github.com/iovisor/bcc/blob/master/tools/memleak_example.txt

```
[…]

98304 bytes in 24 allocations from
stack
        __alloc_pages+0x264 [kernel]
        __alloc_pages+0x264 [kernel]
        alloc_pages_vma+0x7f [kernel]
        do_anonymous_page+0xf4 [kernel]
        __handle_mm_fault+0x8a4
[kernel]
        handle_mm_fault+0xda [kernel]
        do_user_addr_fault+0x1bb
[kernel]
        exc_page_fault+0x7d [kernel]
        asm_exc_page_fault+0x1e
[kernel]
```

Linux Kernel : Memory Management

Kernel Segment / VAS

- 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

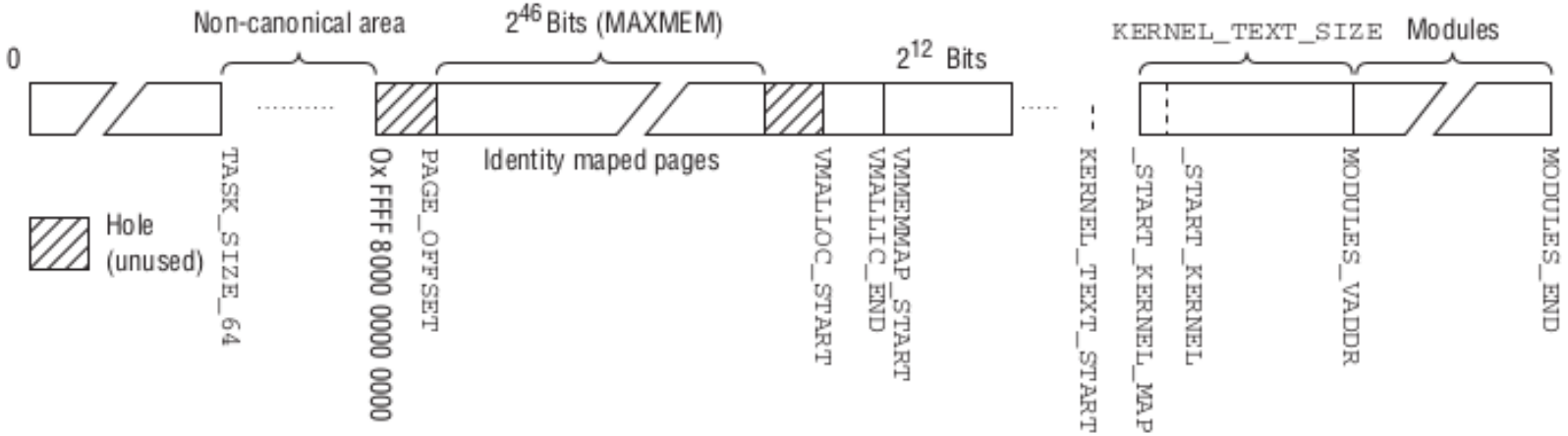
Eg.: On an ARM-32 Linux (Versatile Express CA-9 platform with 512 MB RAM):

```
$ 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)
                                                  << here, 512 MB is the '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 - 0xffff7dfffe00000   (    16 MB)
vmemmap : 0xffff7e0000000000 - 0xffff800000000000   (  2048 GB maximum)
          0xffff7e4d36000000 - 0xffff7e4d36ed0000   (    14 MB actual)
memory  : 0xffff934d80000000 - 0xffff934dbb400000   (   948 MB)
                                  << here, the 'lowmem' region spans 948 MB – the
                                     platform RAM >>
```

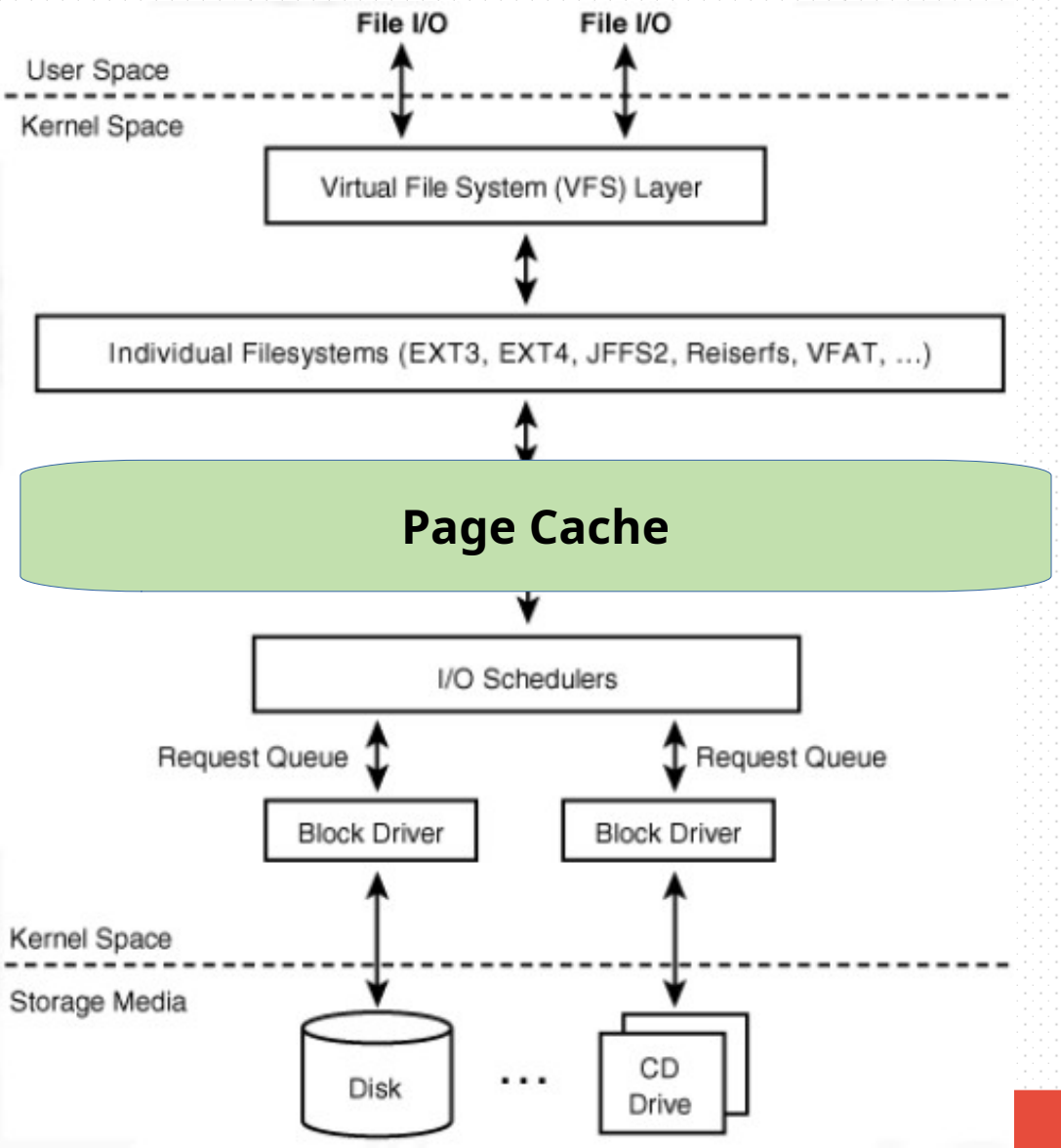
Linux Kernel : Memory Management

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)



Linux Kernel : Memory Management

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
#
```

Quick Tip

Literally watch memory usage change over time with:

```
watch -n 5 -d '/bin/free -m'
```

```
-n : update interval in seconds
-d : Highlight the differences between successive updates...
```

(Src: https://www.redhat.com/sysadmin/one-line-linux-commands)

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.

Linux Kernel : Memory Management

The OOM (Out Of Memory) Killer

- 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

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_**

Linux Kernel : Memory Management

The OOM (Out Of Memory) Killer

- 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*
- *Use choom(1) to query/set the OOM score and/or OOM adjustment value – the 'badness heuristic'*

- Regular user (non-root) can always increase the oom adj value; only as root (or the CAP_SYS_RESOURCE capability) can the OOM adjustment value be lowered

- systemd services can specify the oom_score_adj value within the 'xxx.services' file; examples (under the [Service] section):

```
$ grep -r -I "^OOMScoreAdjust" /lib/systemd/* 2>/dev/null
/lib/systemd/system/systemd-journald.service:OOMScoreAdjust=-250
/lib/systemd/system/systemd-udevd.service:OOMScoreAdjust=-1000
/lib/systemd/system/dbus.service:OOMScoreAdjust=-900
$
```

Also see 'OOMPolicy=' in man systemd.service

Linux Kernel : Memory Management

The OOM (Out Of Memory) Killer

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
[...]
$
```

The OOM (Out Of Memory) Killer and **Demand Paging**

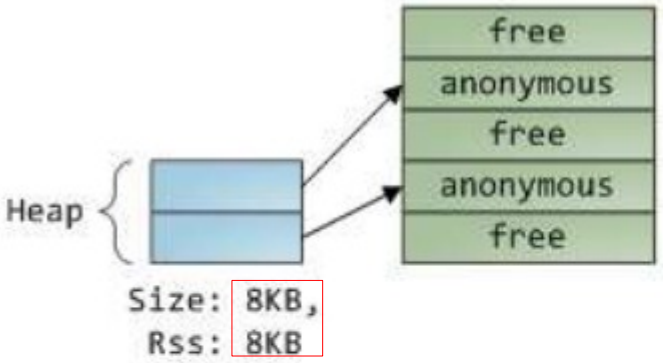
What is Demand Paging?

- When **malloc()** is called, it merely allocates a virtual memory region in the process VAS
- Physical memory page frames are allocated *on-demand*
- *When?*
  - When the page is 'touched' : read / write / execute - on any byte(s) within the page, results in the MMU raising a minor/good page fault
  - the fault handling code determines that the access is legal
  - it asks for a frame, the kernel page allocators allocates one
  - PTEs are stitched up, the faulting access is re-issued and all is well.

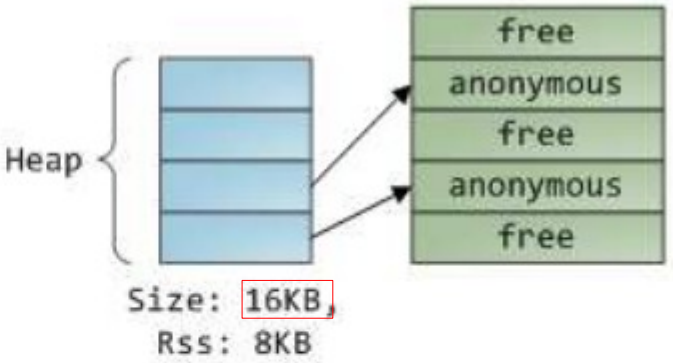
- FYI, the mincore(2) syscall (that's 'm-in-core') can determine whether a given range of pages are resident in memory or not
- (Related: the fincore(1) utility counts the # of pages and size of given file(s) resident in memory)

Linux Kernel : Memory Management

1. Program calls brk() to grow its heap

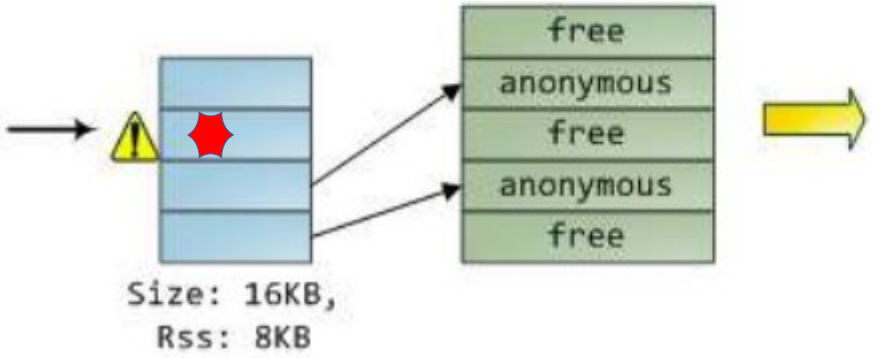


2. brk() enlarges heap VMA.
New pages are **not** mapped onto physical memory.

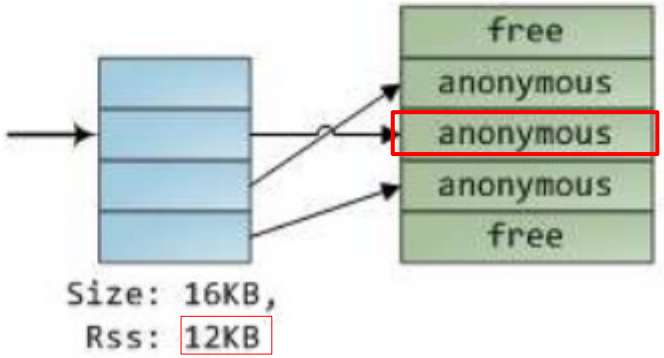


MMU

3. Program tries to access new memory.
Processor page faults.



4. Kernel assigns page frame to process, creates PTE, resumes execution. Program is unaware anything happened.



Src

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*
    - *TIP*- sudo pkill -9 systemd-oomd
      - systemd-oomd can be configured to monitor a given cgroup, and if it's limits are exceeded, kill all processes within it; this can happen before the OOM killer runs… (*link*)

The OOM (Out Of Memory) Killer and Demand Paging

- The output from the kernel log reveals what happened; the OOM killer swooped in & killed the memory hogger process
- here, it's our oom-killer-try process of course (as it most certainly consumes the most memory! see the 'rss' column)
- below, we've shown the output showing mostly processes with negative OOM adjustment values

```
$ dmesg
[ … ]
kernel: Tasks state (memory values in pages):
 kernel: [  pid  ]   uid  tgid total_vm      rss pgtables_bytes swapents oom_score_adj name
 kernel: [    305]     0   305    12948        2   114688      362          -250 systemd-
journal
 kernel: [    337]     0   337     5226        0    65536      800         -1000 systemd-
udevd
 kernel: [    401]   101   401     6055        0    90112     1079             0 systemd-
resolve
                     [ ... ]
 kernel: [    433]   103   433     2098        1    57344      360          -900 dbus-
daemon
 kernel: [    434]     0   434    84060        0   155648      855             0
NetworkManager
                     [ ... ]
 kernel: [    541]     0   541     3071        1    61440      235         -1000 sshd
 kernel: [    550]     0   550     2163        0    57344       40             0 agetty
                     [ ... ]
 kernel: [  13621]  1000 13621   956205   458486  7708672   493339             0
oom_killer_try
 [ … ]
 kernel: oom-
kill:constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=/,mems_allowed=0,global_oom,task_mem
cg=/user.slice/user-1000.slice/session-282.scope,task=oom_killer_try,pid=13621,uid=1000
 kernel: Out of memory: Killed process 13621 (oom_killer_try) total-vm:3824820kB, anon-
```

The OOM (Out Of Memory) Killer and Demand Paging

- We now understand that malloc() merely allocates a virtual memory region in the process VAS
- Physical memory page frames are allocated *on-demand*
- *When?*
  - When the page is 'touched' : read / write / execute - on any byte(s) within the page, results in the MMU raising a minor/good page fault
  - the fault handling code determines that the access is legal
  - it asks for a frame, the kernel page allocators allocates one
  - PTEs are stitched up, the faulting access is re-issued and all is well

  - However:
    - The reality is that we **don't** actually fault on every ***read*** (only on every write/execute)
    - why not?
    - Performance! Instead, the kernel *maps* every new virtual page that's freshly read from, to an underlying *single* kernel frame that's populated with zeroes. So, even if you perform a million reads on a million virtual pages, the kernel ends up physically mapping just one page frame to all of them!
    - helps apps that use a lot of sparse (empty) memory: [K]ASAN, webkit
    - Ref: *What they don't tell you about demand paging in school*, Oct 2020: https://offlinemark.com/2020/10/14/demand-paging/

Linux Kernel : Memory Management

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