Linux Kernel Memory Management

Kernel and Process Segments

Part 2 of 4

Linux Kernel : Memory Management series by kaiwanTECH, © 2000-2015

Part 1 : Introduction to Virtual Memory, Paging

Part 2 : Kernel and Process Segments

Part 3 : Memory Organization

Part 4 : Page Cache, Watermarks, OOM, VMAs

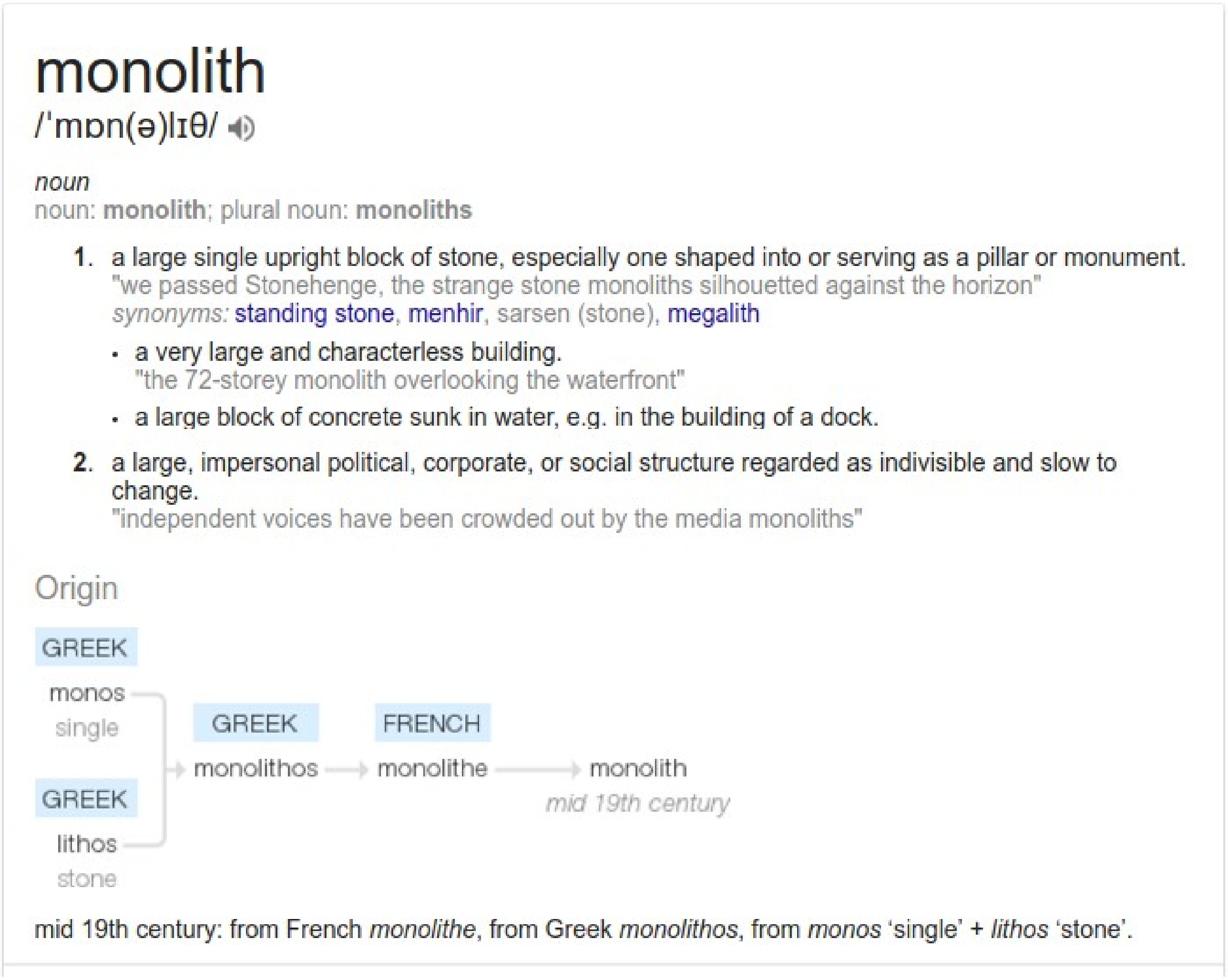
Important Notice : Courseware - Legal

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

Table of Contents

Virtual Address Space Splitting

- Recall that Linux is a **monolithic** OS



monolith
/ˈmɒn(ə)lɪθ/

noun
noun: **monolith**; plural noun: **monoliths**

1. a large single upright block of stone, especially one shaped into or serving as a pillar or monument.
   "we passed Stonehenge, the strange stone monoliths silhouetted against the horizon"
   synonyms: standing stone, menhir, sarsen (stone), megalith
   - a very large and characterless building.
     "the 72-storey monolith overlooking the waterfront"
   - a large block of concrete sunk in water, e.g. in the building of a dock.
2. a large, impersonal political, corporate, or social structure regarded as indivisible and slow to change.
   "independent voices have been crowded out by the media monoliths"

Origin

mid 19th century: from French *monolithe*, from Greek *monolithos*, from *monos* ‘single’ + *lithos* ‘stone’.

- **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]*
    - **On a 64-bit x86-64 and Aarch64 (ARM64) Linux system**
      128 TB : 128 TB :: user-space : kernel-space

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

```
<< Tip:
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)
0x0000 8000 0000 0000 = 128 TB   (2^48)
>>
```

With standard 4 KB Page size

Arch	N-Level	Addr Bits	VM "Split"	Userspace		Kernel-space	
				Start vaddr	End vaddr	Start vaddr	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	128 **PB** : 128 **PB**	0x0	0x007f ffff ffff ffff	0xff80 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 and above*

NOTE!

1. On today's hardware (2015 on) running a modern VM-based OS like Linux, 32-bit microprocessors are slowly being phased out, even in high-end embedded systems (of course, currently several embedded Linux projects continue to use 32-bit ARM or Atom processors; the move to 64-bit is more or less inevitable). This is bound to happen especially with 64-bit processing becoming cheap and an easily available commodity.

This is a good thing: it eliminates many old (but earlier required) workarounds to deal with the limitations of a 32-bit address space. Hence, we now do not focus on "older" features like the 32-bit address space and it's limitations and workarounds - like "high memory", PAE, etc.

***2.** Source *For Security*

```
# Disallow allocating the first 32k of memory (cannot be 64k due to ARM loader).
CONFIG_DEFAULT_MMAP_MIN_ADDR=32768
# For maximal userspace memory area (and maximum ASLR).
CONFIG_VMSPLIT_3G=y
```

3. It *is* possible to avoid splitting the VAS altogether; to allow the use of the entire virtual address-space for each of user and kernel mode of a process. Doing this efficiently however, is difficult, and requires underlying hardware support (ASI identifiers on the UltraSparc, ..). The reality is it's generally not worth it, hence the VM split option is the one used.

4. Aarch64 Linux memory layout

5. *[NEW!]* *[Dec 2017, announced 03 Jan 2018]*
The Meltdown and Spectre Security Bugs!

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… See this *LWN article: "KAISER: hiding the kernel from user space"*.

[KAISER: Kernel Address Isolation to have Side-channels Efficiently Removed. Renamed (?) to KPTI: Kernel Page Table Isolation].

Wikipedia :: Meltdown (security vulnerability)

Source: meltdownattack
…
Meltdown and Spectre exploit critical vulnerabilities in modern . These hardware bugs allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. This might include your passwords stored in a password manager or browser, your personal

photos, emails, instant messages and even business-critical documents.
…
Meltdown breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus also the secrets, of other programs and the operating system.

If your computer has a vulnerable processor and runs an unpatched operating system, it is not safe to work with sensitive information without the chance of leaking the information. This applies both to personal computers as well as cloud infrastructure. Luckily, there are software patches against Meltdown.

…

KPTI, LWN, 20 Dec 2017, Corbet

… 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. ...

```
$ cat /etc/fedora-release
Fedora release 27 (Twenty Seven)
$ 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
$
```

.. And on the “Seawolf” Ubuntu-based VM:

```
$ uname -a
Linux seawolf-mindev 4.13.0-36-generic #40-Ubuntu SMP Fri Feb 16 20:07:48 UTC
2018 x86_64 x86_64 x86_64 GNU/Linux
$ dmesg |grep isolation
[    0.000000] Kernel/User page tables isolation: enabled
$ jlog |grep -i spectre
Mar 04 17:39:33 seawolf-mindev kernel: Spectre V2 mitigation: Mitigation:
Full generic retpoline
Mar 04 17:39:33 seawolf-mindev kernel: Spectre V2 mitigation: Speculation
```

```
control IBPB not-supported IBRS not-supported
Mar 04 17:39:33 seawolf-mindev kernel: Spectre V2 mitigation: Filling RSB on
context switch
$
```

Pass 'nopti' to disable KPTI.

Also (on sufficiently recent kernels):

```
$ ls /sys/devices/system/cpu/vulnerabilities/
meltdown  spectre_v1  spectre_v2
$ cat /sys/devices/system/cpu/vulnerabilities/*
Mitigation: PTI
Mitigation: __user pointer sanitization
Mitigation: Full generic retpoline, IBPB
$
```

Additionally:
Reported (and detailed explanation) by Google's Project Zero

Dave Hansen's kernel notes on the overhead: https://lkml.org/lkml/2018/1/4/775

Papers (PDF)
https://meltdownattack.com/meltdown.pdf
https://spectreattack.com/spectre.pdf

Which CPUs are affected?
https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability)#Affected_hardware
ARM :: *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism*

Meltdown, Spectre: The password theft bugs at the heart of Intel CPUs, The Register, 4 Jan 2018

KPTI/KAISER Meltdown Initial Performance Regressions, Brendan Gregg

The mysterious case of the Linux Page Table Isolation patches
Spectre & Meltdown: tapping into the CPU's subconscious thoughts

… and

Name	*CVE ID [on cvedetails]*	*Product*	*Date*	*Vuln Type*	*CVSS Score*	*See More here*
Meltdown & Spectre	CVE-2017	Hardware Microprocessors from	04-Jan-2018	Microprocessor out-of-order speculative	4.7	• CVE-2017-5753: Known as Variant 1, a bounds check bypass

	- 5753	Intel, AMD, ARM !		execution of instructions		• CVE-2017-5715: Known as Variant 2, branch target injection • CVE-2017-5754: Known as Variant 3, rogue data cache load.

<<
NEW [29Oct2019]

Running on Intel? If you want security, disable hyper-threading, says Linux kernel maintainerSpeculative execution bugs will be with us for a very long time >>

<< More coverage on Meltdown/Spectre at end of this module >>

*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.*

<<
Source: "Professional Linux Kernel Architecture", W Mauerer, Wrox Press

IA-32 systems the kernel typically divides the total available virtual address space of 4 GiB in a ratio of 3 : 1.

The lower 3 GiB are available for user-mode applications, and the upper gigabyte is reserved exclusively for the kernel. Whereas the current system context is irrelevant when assigning the virtual address space of the kernel, each process has its own specific
address space.

The major reasons for this division are as follows:

❑ When execution of a user application switches to kernel mode (this always happens when, e.g., a system call is used or a periodic timer interrupt is generated), the kernel must be embedded in a reliable environment. It is therefore essential to assign part of the address space exclusively to the kernel.

❑ The physical pages (RAM) are mapped to the start of the kernel address space so that the kernel can access them directly without the need for complicated page table operations.

<<
Source
"… this mechanism persists for a simple reason: getting rid of it would make the system run considerably slower. Keeping the kernel permanently mapped eliminates the need to flush the processor's translation lookaside buffer (TLB) when switching between user and kernel space, and it allows the TLB entries for kernel space to never be flushed. Flushing the TLB is an expensive operation for a couple of reasons: having to go to the page tables to repopulate the TLB hurts, but the act of performing the flush itself is slow enough that it can be the biggest part of the cost. ..."
>>

If all physical pages were mapped into the address space accessible to userspace processes, this would lead to serious security problems if several applications were running on the system. Each application would then be able to read and modify the memory areas of other processes in physical RAM. Obviously this must be prevented at all costs.

While the virtual address portion employed for userland processes changes with every task switch, the kernel portion is always the same. The situation is summarized in Figure 3-14.

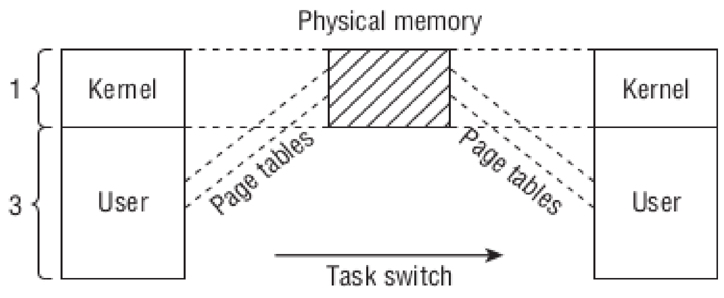


Figure 3-14: Connection between virtual and physical address space on IA-32 processors.
…
>>

- In User Mode, a process can only reference virtual addresses upto PAGE_OFFSET-1; addresses above this can only be referenced when the process executes in Kernel Mode

- Obviously, the entries of the PGD of a regular user process that map it's User Mode address space varies depending on the process

- The PGD entries for a process for addresses >= PAGE_OFFSET map to the kernel master PGD and are the *same for all user processes*

Useful Resource: Anatomy of a Program in Memory

- U*pon boot, the Linux kernel* performs *an identity-mapping (1:1 mapping) of physical RAM to kernel virtual address space:*

Physical RAM : Kernel space (upto the ZONE_NORMAL extent)

- All available physical memory (RAM) is mapped into the kernel address space. Given that the size of the kernel (virtual) address space is 1 GB (on a 32-bit processor), this implies that physical memory within 1 GB is directly mapped into this space and therefore, in practice, address translation is very quick:

physical address = virtual address – 3GB offset ;

More correctly,
phy addr = virt addr - PAGE_OFFSET

(Careful! This does *not* always work; it only works for “direct-mapped” or “low” memory. More details in the Appendices).

- This scheme works well for RAM upto the limit (actually 896MB on IA-32); any physical RAM above this necessarily needs to be mapped using the *highmem* method.

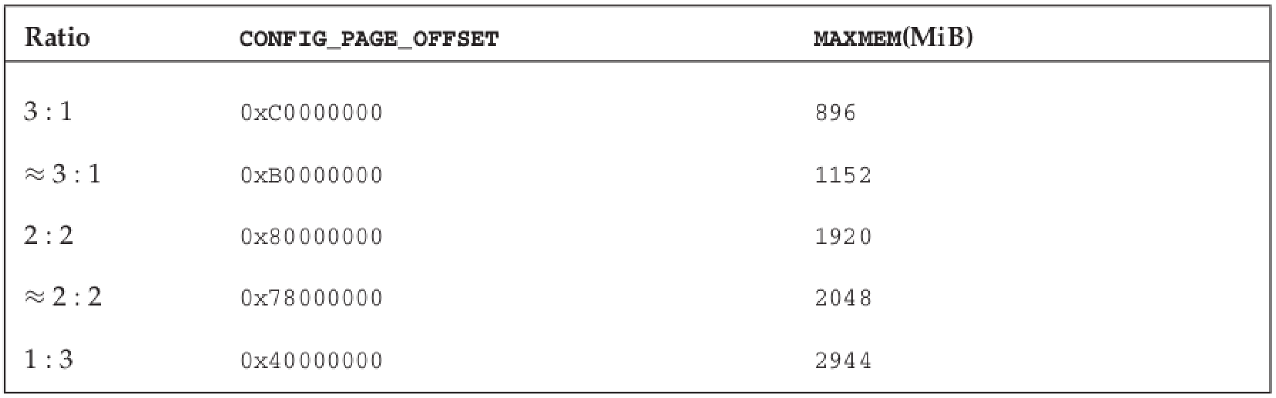
- With highmem, in order to access those page frames above the limit (say 896 MB), the kernel has to resort to mapping (and then unmapping) them into and out of kernel virtual address space (using kmap|kunmap).

- It's important to understand that these considerations (highmem or not) never arise for userspace processes' pages: they're *always* accessed using page tables.

Why use different “splitting” ratios?
[*Source*, *PLKA*] See the table below:

User : Kernel virtual address space

Table 3-6: Different Splitting Ratios for the IA-32 Virtual Address Space, and the Resulting Maximum Identity-Mapped Physical Memory.



Ratio	CONFIG_PAGE_OFFSET	MAXMEM(MiB)
3 : 1	0xC0000000	896
≈ 3 : 1	0xB0000000	1152
2 : 2	0x80000000	1920
≈ 2 : 2	0x78000000	2048
1 : 3	0x40000000	2944

Decreasing the userland virtual-address space (VAS) has the beneficial side effect of allowing us *to directly map* in more physical platform RAM; but of course, at the loss of VAS for userland.

SIDEBAR :: Q&A on Quora

Does the Linux kernel have its own virtual address space like every user mode process does?

1. Does the kernel have any virtual memory at all ?

2. How do the kernel developers avoid the cost of context switch between address spaces on every system call ?

Robert Love, I hack on the Linux kernel.

48 upvotes by Don Marti, Lorin Ricker, Vrushali Kulkarni, (more)

Yes, the Linux kernel uses virtual memory just as user-space processes use virtual memory. That virtual memory is special in some ways—the kernel controls it, after all—but it is virtual, not physical.

Linux avoids the cost of switching out the memory address space on each entry to and exit from the kernel by mapping the kernel's virtual memory address into each process's virtual address space. The process thus shares its virtual address space with the kernel, obviating the need to perform any switch when entering or exiting kernel-space.

On x86-32, for example, the kernel occupies the first 1GB of virtual address space while the process occupies the remaining 3GB. The kernel uses memory permissions so that the user process cannot read from or write to its 1GB.

Splitting the address space like this has the upside of avoiding an address space swap on every context switch but the downside of limiting the kernel and user-space to less than the full 4GB that a 32-bit system would afford. The tradeoff is generally worth it. On a 64-bit system with copious virtual address space, the downside is nearly zero.

Q. Are kernel virtual addresses, especially those that are direct-mapped, translated via the kernel paging table (or just as an offset calculation)?

A. Once paging is enabled, all virtual addresses get translated to physical addresses via paging tables. Direct-mapped kernel virtual address space just being an offset from physical RAM makes the *virt_to_phys()* and *phys_to_virt()* macros easy to implement (just a subtraction / addition (which can also be efficiently implememted by bit-shifting)).

Precisely because much of the kernel address space is direct-mapped and remains that way for the life of the system (think the kernel code/ text/bss and kmalloc regions), the page tables for that purpose are setup at boot time and never need to be changed! (It's only the dynamic (de)allocations (like [k|v]malloc (/ioremap) and DMA kernel mappings) that need to be dynamically setup and maintained).

The fact remains that they are translated via the kernel master paging table (*swapper_pg_dir* - which incidentally is in the "swapper" task – the cpu idle thread pid 0).

Every user-mode process, when it switches to kernel mode (syscall or interrupt), maps into the same kernel-mode paging tables. Hence kernel virtual addresses are used when in kernel mode, and are translated to physical (DRAM) addresses via the TLB or kernel paging tables.

The reasons that kernel-space address translation remains, in reality, very fast:

- the OS will typically exploit the microprocessor and pin down the TLBs for kernel-space
- TLBs on modern processors are large-ish
- the kernel will try and use huge pages (2 MB on x86_64) when enabled and possible (reducing TLB pressure even further. Via the Transparent Huge Pages (THP) feature).

Your author asked a question on StackOverflow regarding the translation of kernel virtual addresses to physical addresses. Please do read it and the 'correct' response.

How exactly do kernel virtual addresses get translated to physical RAM?

What is THP ?

THP- Transparent Huge Pages:

Source

Recommended LWN article: Transparent huge pages in 2.6.38

Processors manage memory in small units called "pages" (which is 4 KB in size in x86). Each process has a virtual memory address space, and there is a "page table" where all the correspondencies between each virtual memory address page and its correspondent real RAM page are kept.

The work of walking the page table to find out which RAM page corresponds to a given virtual address is expensive, so the CPU has a small cache << the TLB >> to store the result of that work for frequently accessed virtual addresses. However, this cache is not very big and it only supports 4KB pages, so many data-intensive workloads (databases, KVM) have performance problems because all their frequently accessed virtual addresses can't be cached.

To solve this problem, modern processors add cache entries that support pages bigger than 4KB (like 2MB/4MB). Until now, the one way that userspace had to use those pages in Linux was hugetblfs, a filesystem-based API.

This release adds support for transparent hugepages (- hugepages are used automatically where possible. Transparent Huge Pages can be configured to be used always or only as requested with madvise(MADV_HUGEPAGE), and its behaviour can be changed online in */sys/kernel/mm/transparent_hugepage/enabled*. For more details, check Documentation/vm/transhuge.txt

From “Transparent huge pages in 2.6.38”, LWN*:*
…
Huge pages can improve performance through reduced page faults (a single fault brings in a large chunk of memory at once) and by reducing the cost of virtual to physical address translation (fewer levels of page tables must be traversed to get to the physical address). But the real advantage comes from avoiding translations altogether. If the processor must translate a virtual address, it must go through as many as four levels of page tables, each of which has a good chance of being cache-cold, and, thus, slow. For this reason, processors maintain a "translation lookaside buffer" (TLB) to cache the results of translations.
...”

From madvise(2) man page:
…
MADV_HUGEPAGE (since Linux 2.6.38)
Enables Transparent Huge Pages (THP) for pages in the range specified by addr and length. Currently, Transparent Huge Pages work only with private anonymous pages (see mmap(2)).

The kernel will regularly scan the areas marked as huge page candidates to replace them with huge pages. The kernel will also allocate huge pages directly when the region is naturally aligned to the huge page size (see posix_memalign(2)). This feature is primarily aimed at applications that use large mappings of data and access large regions of that memory at a time (e.g., virtualization systems such as QEMU).

It can very easily waste memory (e.g., a 2MB mapping that only ever accesses 1 byte will result in 2MB of wired memory instead of one 4KB page). See the Linux kernel source file *Documentation/vm/transhuge.txt* for more details. The MADV_HUGEPAGE and MADV_NOHUGEPAGE operations are available only if the kernel was configured with CONFIG_TRANSPARENT_HUGEPAGE.
…

THP: 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)

* Earlier, for applications to take advantage of THP was not easy: it required significant work from app developers and system administrators, taking advantage of the HugeTLBFS feature. On modern kernels, it's much easier- apps are completely transparent to THP; the kernel will take advantage when it can.

Also, from the kernel documentation kernel-parameters.txt:

```
"…
hugepages=    [HW,X86-32,IA-64] HugeTLB pages to allocate at boot.
hugepagesz=   [HW,IA-64,PPC,X86-64] The size of the HugeTLB pages.
                      On x86-64 and powerpc, this option can be specified
                      multiple times interleaved with hugepages= to reserve
                      huge pages of different sizes. Valid pages sizes on
                      x86-64 are 2M (when the CPU supports "pse") and 1G
                      (when the CPU supports the "pdpe1gb" cpuinfo flag).
..."
```

Ref:
How to use, monitor, and disable transparent hugepages in Red Hat Enterprise Linux 6?
…
Check whether THP is in use:

```
# grep -i AnonHugePages /proc/meminfo
AnonHugePages:    747520 kB
#
```

More details:

```
# egrep 'trans|thp' /proc/vmstat
nr_anon_transparent_hugepages 365
thp_fault_alloc 7435
thp_fault_fallback 1333
thp_collapse_alloc 3497
thp_collapse_alloc_failed 342
thp_split 3553thp_zero_page_alloc 4
thp_zero_page_alloc_failed 0
#
```

<< Above values explained in the THP documentation >>

Check THP usage per process?

```
sudo grep -e AnonHugePages /proc/*/smaps | awk '{ if($2>4) print $0}' | awk -F "/" '{
print $0; system("ps -fp " $3)}'

Eg. output:
...
/proc/10093/smaps:AnonHugePages:     26624 kB
UID        PID  PPID  C STIME TTY          TIME CMD
root     10093 10092  4 06:27 pts/7    00:00:13 qemu-system-x86_64 --enable-kvm -
kernel images/bzImage -drive file=images/wheezy.img,if=virtio,format=r..
...
/proc/11839/smaps:AnonHugePages:      4096 kB
UID        PID  PPID  C STIME TTY          TIME CMD
kaiwan   11839  2328  0 Mar28 ?        00:05:04 /opt/google/chrome/chrome --
type=renderer --enable-features=LinuxObsoleteSystemIsEndOfTheLine<LinuxObso
/proc/15485/smaps:AnonHugePages:      2048 kB
UID        PID  PPID  C STIME TTY          TIME CMD
kaiwan   15485  1415  0 Mar29 ?        00:09:05 /usr/lib/firefox/firefox
...
/proc/8122/smaps:AnonHugePages:     28672 kB
UID        PID  PPID  C STIME TTY          TIME CMD
kaiwan    8122  8104  0 Mar27 ?        00:08:29
/usr/lib/libreoffice/program/soffice.bin --writer file:/
...
/proc/9780/smaps:AnonHugePages:     26624 kB
UID        PID  PPID  C STIME TTY          TIME CMD
kaiwan    9780  1415  0 06:09 ?        00:00:01 evince <...>/git_basics.pdf
...
```

Enable/Disable THP at boot:

```
…
transparent_hugepage=
                     [KNL]
                     Format: [always|madvise|never]
                     Can be used to control the default behavior of the system
                     with respect to transparent hugepages.
                     See Documentation/vm/transhuge.txt for more details.
...


At any time:
echo always >/sys/kernel/mm/transparent_hugepage/enabled
echo madvise >/sys/kernel/mm/transparent_hugepage/enabled
echo never >/sys/kernel/mm/transparent_hugepage/enabled
```

Imp: see the THP latest documentation.

64-bit Linux (on the x86-64 architecture)

- Allows up to **128 TB** (1 TB = 1000 GB) of address space for individual processes, and can address approximately 64 TB of physical memory, subject to processor and system limitations.

- **The default VM split is 128 TB : 128 TB :: user-space : kernel-space (!)**

  - "Canonical form addresses run from
    0x0 through 0x00007FFF`FFFFFFFF <<user-space>>, and from
    0xFFFF8000`00000000 through 0xFFFFFFFF`FFFFFFFF <<kernel-space>>,
    for a total of 256 TB of usable virtual address space."

- The amount of physical RAM that can be used is currently limited to about 64 TB (depends on processor and system limitations).

<< A quick check: on an x86_64 Linux system, look up the kernel core image (snapshot):

```
# ls -lh /proc/kcore
-r-------- 1 root root 128T Apr  3 07:08 /proc/kcore
#
```

>>

Source

…

64-bit systems allow addressing 2 to the 64th power bytes of data in theory, but no current chips allow accessing all 16 exabytes (18,446,744,073,709,551,616 bytes).

For example, AMD architecture uses only the lower 48 bits of an address, and bits 48 through 63 must be a copy of bit 47 or the processor raises an exception. Thus addresses are 0 through 00007FFF`FFFFFFFF, and from FFFF8000`00000000 through FFFFFFFF`FFFFFFFF, for a total of 256 TB (281,474,976,710,656 bytes) of usable virtual address space. Another downside is that addressing all 64 bits of memory requires a lot more paging tables for the OS to store, using valuable memory for systems with less than all 16 exabytes installed. Note these are virtual addresses, not physical addresses.

As a result, many operating systems use the higher half of this space for the OS, starting at the top and growing down, while user programs use the lower half, starting at the bottom and growing upwards. Current Windows* versions use 44 bits of addressing (16 terabytes = 17,592,186,044,416 bytes). The resulting addressing is shown in Figure 2. The resulting addresses are not too important for user programs since addresses are assigned by the OS, but the distinction between user addresses and kernel addresses are useful for debugging.

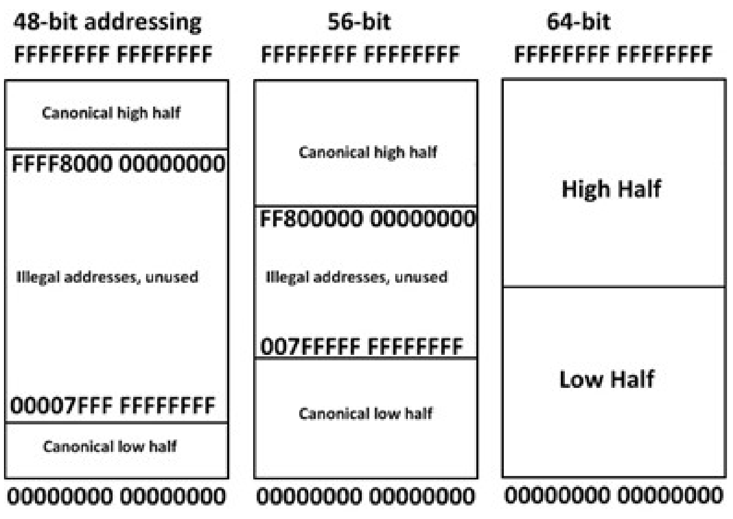


Figure 2 - *Memory Addressing*

- More Details:

Source (below): http://en.wikipedia.org/wiki/X86-64#Virtual_address_space_details

…

Canonical form addresses

Although virtual addresses are 64 bits wide in 64-bit mode, current implementations (and any chips known to be in the planning stages) do not allow the entire virtual address space of 264 bytes (16 EB) to be used.

<<

Source: AMD64 Architecture Programmer's Manual, Vol 2: Systems Programming

...

Long mode defines 64 bits of virtual-address space, but processor implementations can support less. Although some processor implementations do not use all 64 bits of the virtual address, they all check bits 63 through the most-significant implemented bit to see if those bits are all zeros or all ones. An address that complies with this property is in canonical address form. In most cases, a virtual-memory reference that is not in canonical form causes a general-protection exception (#GP) to occur.

>>

Most operating systems and applications will not need such a large address space for the foreseeable future (for example, Windows implementations for AMD64 are only populating 16 TB, or 44 bits' worth), so implementing such wide virtual addresses would simply increase the complexity and cost of address translation with no real benefit.

AMD therefore decided that, in the first implementations of the architecture, only the least significant 48 bits of a virtual address would actually be used in address translation (page table lookup).[1](p130) Further, bits 48 through 63 of any virtual address must be copies of bit 47 (in a manner akin to sign extension), or the processor will raise an exception. Addresses complying with this rule are referred to as "canonical form."[1](p128)

Canonical form addresses run from 0 through 00007FFF`FFFFFFFF, and from FFFF8000`00000000 through FFFFFFFF`FFFFFFFF, for a total of 256 TB of usable virtual address space.

This "quirk" allows an important feature for later scalability to true 64-bit addressing: many operating systems (including, but not limited to, the Windows NT family) take the higher-addressed half of the address space (named kernel space) for themselves and leave the lower-addressed half (user space) for application code, user mode stacks, heaps, and other data regions.

The "canonical address" design ensures that every AMD64 compliant implementation has, in effect, two memory halves: the lower half starts at 00000000`00000000 and "grows upwards" as more virtual address bits become available, while the higher half is "docked" to the top of the address space and grows downwards. Also, fixing the contents of the unused address bits prevents their use by operating system as flags, privilege markers, etc., as such use could become problematic when the architecture is extended to 52, 56, 60 and 64 bits.

…

[Source: http://en.wikipedia.org/wiki/Terabyte]

SI decimal prefixes		Binary usage	IEC binary prefixes		
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}	
See also: Multiples of bits · Orders of magnitude of data					

x86_64 Architecture Canonical Addressing

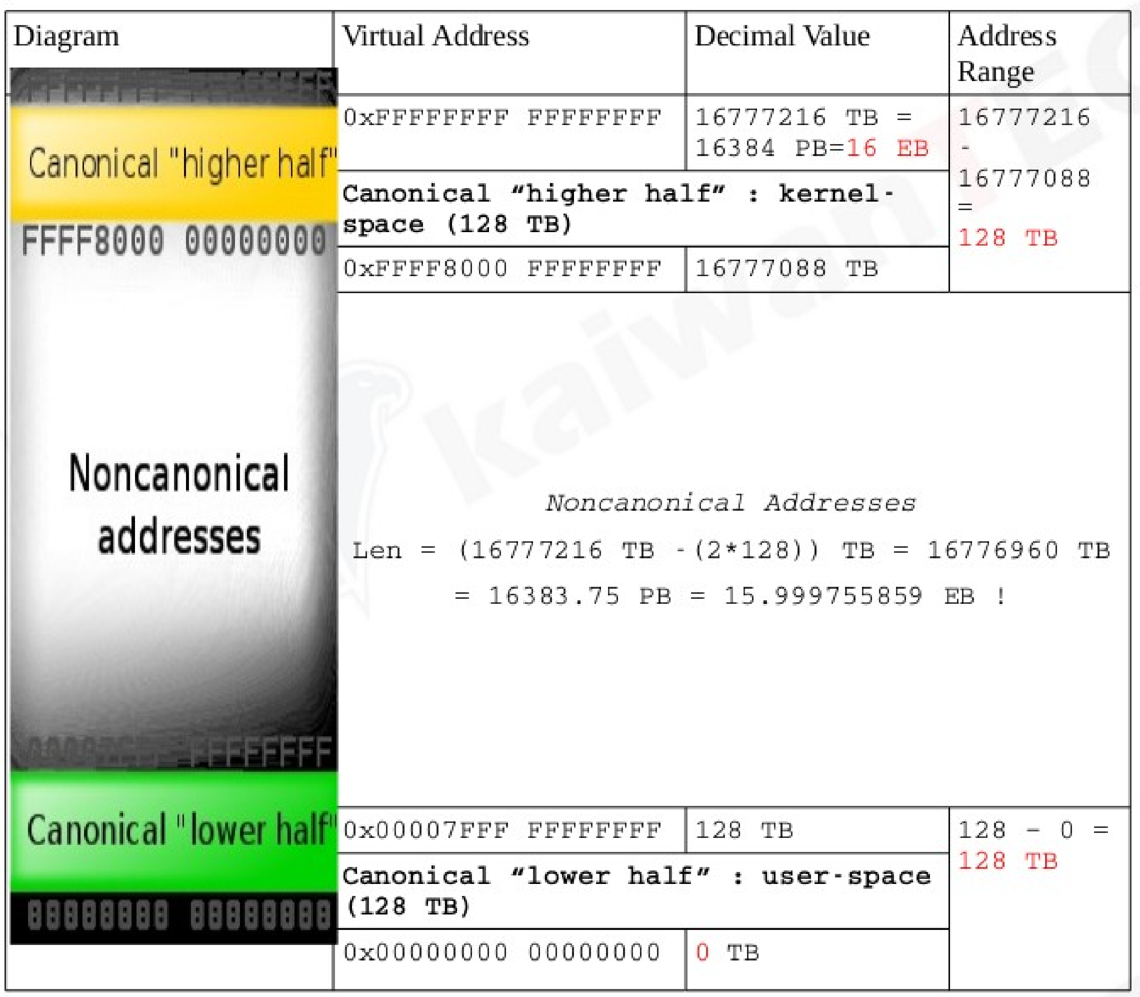

Canonical "higher half"
FFFF8000 00000000
Noncanonical addresses
FFFFFFFF
Canonical "lower half"
00000000 00000000

Diagram	Virtual Address	Decimal Value	Address Range
	0xFFFFFFFF FFFFFFFF	16777216 TB = 16384 PB=16 EB	16777216 - 16777088 = 128 TB
	Canonical "higher half" : kernel-space (128 TB)		
	0xFFFF8000 FFFFFFFF	16777088 TB	
	Noncanonical Addresses Len = (16777216 TB -(2*128)) TB = 16776960 TB = 16383.75 PB = 15.999755859 EB !		
	0x00007FFF FFFFFFFF	128 TB	128 – 0 = 128 TB
	Canonical "lower half" : user-space (128 TB)		
	0x00000000 00000000	0 TB	

<< Typo: kernel lower addres is 0xffff 8000 0000 0000 not 0xffff 8000 ffff ffff as shown >>

<< See notes below for the VM layout on the Aarch64 >>

A Quick Examination of the Process Usermode VAS (Virtual Address Space)

```
$ cat vm_user.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

typedef unsigned int u32;
typedef long unsigned int u64;

int g=5, u;

int main(int argc, char **argv)
{
	int local;
	char *heapptr = malloc(100);

	// OS bits
	// for cpu use 'lscpu'
	printf("wordsize : %d\n", __WORDSIZE);

	if (__WORDSIZE == 32) {
		printf("&main = 0x%08x &g = 0x%08x &u = %08x &heapptr = 0x%08x &loc = 0x
%08x\n",
			(u32)main, (u32)&g, (u32)&u, (u32)heapptr, (u32)&local);
	} else if (__WORDSIZE == 64) {
		printf("&main = 0x%016lx &g = 0x%016lx &u = %016lx &heapptr = 0x%016lx
&loc = 0x%016lx\n",
			(u64)main, (u64)&g, (u64)&u, (u64)heapptr, (u64)&local);
	}

	free(heapptr);
	return 0;
}
$ gcc vm_user.c -Wall -o vm_user
$
```

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 = 0x00000000004005bd &gi = 0x0000000000601050 &gu = 0x0000000000601058 &heapptr
= 0x0000000001c8d010 &loc = 0x00007fff918377b4
$
```

Running on a 64-bit (Aarch64 Raspberry Pi 3 Model B+) system (and OS) [256 TB : 256 TB VM split]

```
rpi64 $ ./vm_user
wordsize : 64
&main = 0x0000aaaad3d708a4 &g = 0x0000aaaad3d81010 &u = 0x0000aaaad3d81018 &heapptr =
0x0000aaab0958c260 &loc = 0x0000ffffda6d8f8c
rpi64 $
```

The value `0x0000ffffda6d8f8c` seen above (the top of the usermode stack of this process) in decimal is (very close to) 256 TB!

Aarch64 (ARM64)

Ref: *Documentation/arm64/memory.txt*

```
"...
This document describes the virtual memory layout used by the AArch64
Linux kernel. The architecture allows up to 4 levels of translation
tables with a 4KB page size and up to 3 levels with a 64KB page size.

AArch64 Linux uses either 3 levels or 4 levels of translation tables
with the 4KB page configuration, allowing 39-bit (512GB) or 48-bit
(256TB) virtual addresses, respectively, for both user and kernel. With
64KB pages, only 2 levels of translation tables, allowing 42-bit (4TB)
virtual address, are used but the memory layout is the same.

User addresses have bits 63:48 set to 0 while the kernel addresses have
the same bits set to 1. TTBRx selection is given by bit 63 of the
virtual address << hence TTBR1 used to point to the kernel masted PGD -
swapper_pg_dir - while TTBR0 points to each process' individual PGD at runtime >>.
The swapper_pg_dir contains only kernel (global) mappings while the user pgd contains
only user (non-global) mappings. The swapper_pg_dir address is written to TTBR1 and
never written to TTBR0.


AArch64 Linux memory layout with 4KB pages + 3 levels:

Start                  End                    Size          Use
```

```
-----------------------------------------------------------------------
0000000000000000   0000007fffffffff    512GB       user
ffffff8000000000   ffffffffffffffff    512GB       kernel


AArch64 Linux memory layout with 4KB pages + 4 levels:

Start                End                 Size         Use
-----------------------------------------------------------------------
0000000000000000   0000ffffffffffff    256TB       user
ffff000000000000   ffffffffffffffff    256TB       kernel

...

Translation table lookup with 4KB pages:

+--------+--------+--------+--------+--------+--------+--------+--------+
|63    56|55    48|47    40|39    32|31    24|23    16|15     8|7      0|
+--------+--------+--------+--------+--------+--------+--------+--------+
 |                 |         |         |         |         |
 |                 |         |         |         |         v
 |                 |         |         |         |   [11:0]  in-page offset
 |                 |         |         |         +-> [20:12] L3 index
 |                 |         |         +-----------> [29:21] L2 index
 |                 |         +---------------------> [38:30] L1 index
 |                 +-------------------------------> [47:39] L0 index
 +-------------------------------------------------> [63] TTBR0/1

...“
```

<<

Additional / FYI

Mitigating the Performance Impact of Meltdown/Spectre

Based upon: http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html

“… The KPTI patches to mitigate Meltdown can incur massive overhead, anything from 1% to over 800%. Where you are on that spectrum depends on your syscall and page fault rates, due to the extra CPU cycle overheads, and your memory working set size, due to TLB flushing on syscalls and context switches. ...”

Recommendations

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

>>

<<
Meltdown: What exactly is the problem?

https://arstechnica.com/gadgets/2018/01/heres-how-and-why-the-spectre-and-meltdown-patches-will-hurt-performance/

…
Meltdown applies to Intel's x86 and Apple's ARM processors; it will also apply to ARM processors built on the new A75 design. Meltdown is fixed by changing how operating systems handle memory. Operating systems use structures called page tables to map between process or kernel memory and the underlying physical memory. Traditionally, the accessible memory given to each process is split in half; the bottom half, with a per-process page table, belongs to the process. The top half belongs to the kernel. This kernel half is shared between every process, using just one set of page table entries for every process. This design is both efficient—the processor has a special cache for page table entries—and convenient, as it makes communication between the kernel and process straightforward.

The fix for Meltdown is to split this shared address space. That way when user programs are running, the kernel half has an empty page table rather than the regular kernel page table. This makes it impossible for programs to speculatively use kernel addresses.

…

The special cache used for page table entries, called the translation lookaside buffer (TLB), is an important and limited resource that contains mappings from virtual addresses to physical memory addresses. Traditionally, the TLB gets flushed—emptied out—every time the operating system switches to a different set of page tables. This is why the split address was so useful; switching *from* a user process *to* the kernel could be done without having to switch to a different set of page tables (because the top half of each user process is the shared kernel page table). Only switching from one user process to a *different* user process requires a change of page tables (to switch the bottom half from one process to the next).

The dual page table solution to Meltdown increases the number of switches, forcing the TLB to be flushed not just when switching from one user process to the next, but also when one user process calls into the kernel. Before dual page tables, a user process that called into the kernel and then received a response wouldn't need to flush the TLB at all, as the entire operation could use the same page table. Now, there's one page table switch on the way into the kernel, and a second, back to the process' page table, on the way out. This is why I/O intensive workloads are penalized so heavily: these workloads switch from the benchmark process into the kernel and then back into the benchmark process over and over again, incurring two TLB flushes for each roundtrip.

…

<*Wrt PCID: Intel's Process Context Identifiers*>
This makes a difference. In a synthetic benchmark that tests only the cost of switching into the kernel and back again, an unpatched Linux system can switch about 5.2 million times a second. Dual page tables slashes that to 2.2 million a second; dual page tables with PCID gets it back up to 3 million.
…

<< End of Linux Memory Management, Part 2 >>

kaiwa
Please *rns,* *techni*