Linux Kernel Memory Management

Memory Organization, Kernel Segment

Part 3 of 4

Linux Kernel : Memory Management series by kaiwanTECH

Part 1 : Introduction to Virtual Memory, Paging

Part 2 : Kernel and Process Segments

Part 3 : Memory Organization, Kernel Segment

Part 4 : Page Cache, Watermarks, OOM, VMAs

Important Notice : Courseware - Legal

Table of Contents

Memory Organization

[Source](#) (below diagram): Computer Architecture, A Quantitative Approach, 5th Ed by Hennessy & Patterson.

Symmetric Multi Processing (SMP) Architecture

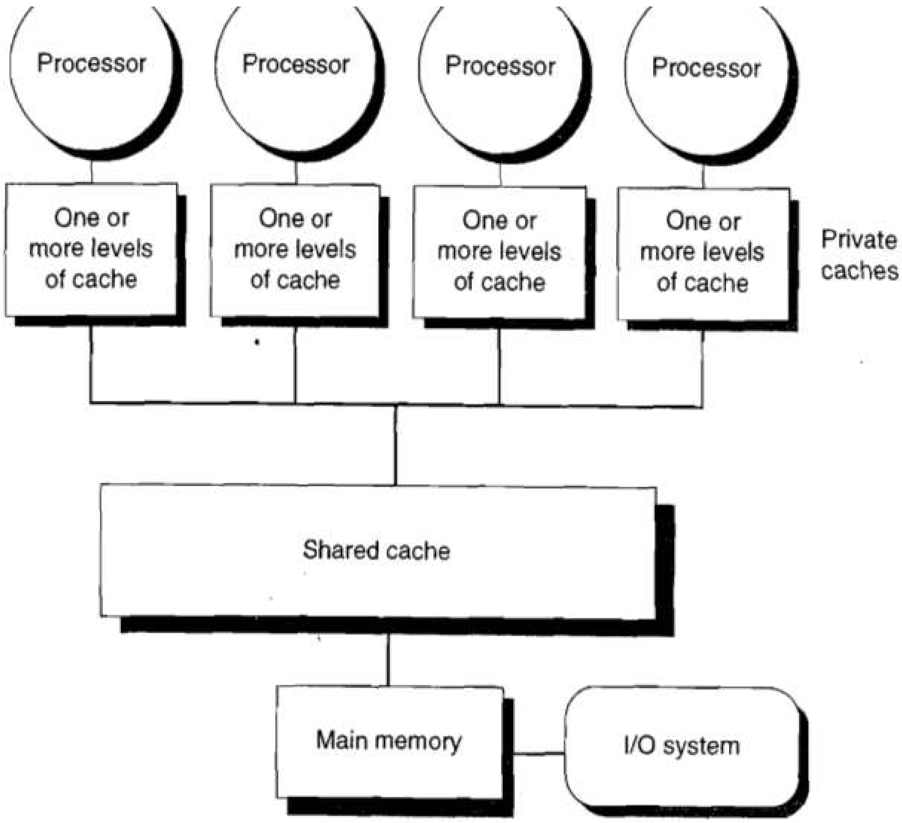


Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor–cache subsystems share the same physical memory, typically with one level of shared cache, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip version the shared cache would be omitted and the bus or interconnection network connecting the processors to memory would run between chips as opposed to within a single chip.

Distributed Shared Memory (DSM -or- NUMA) Architecture

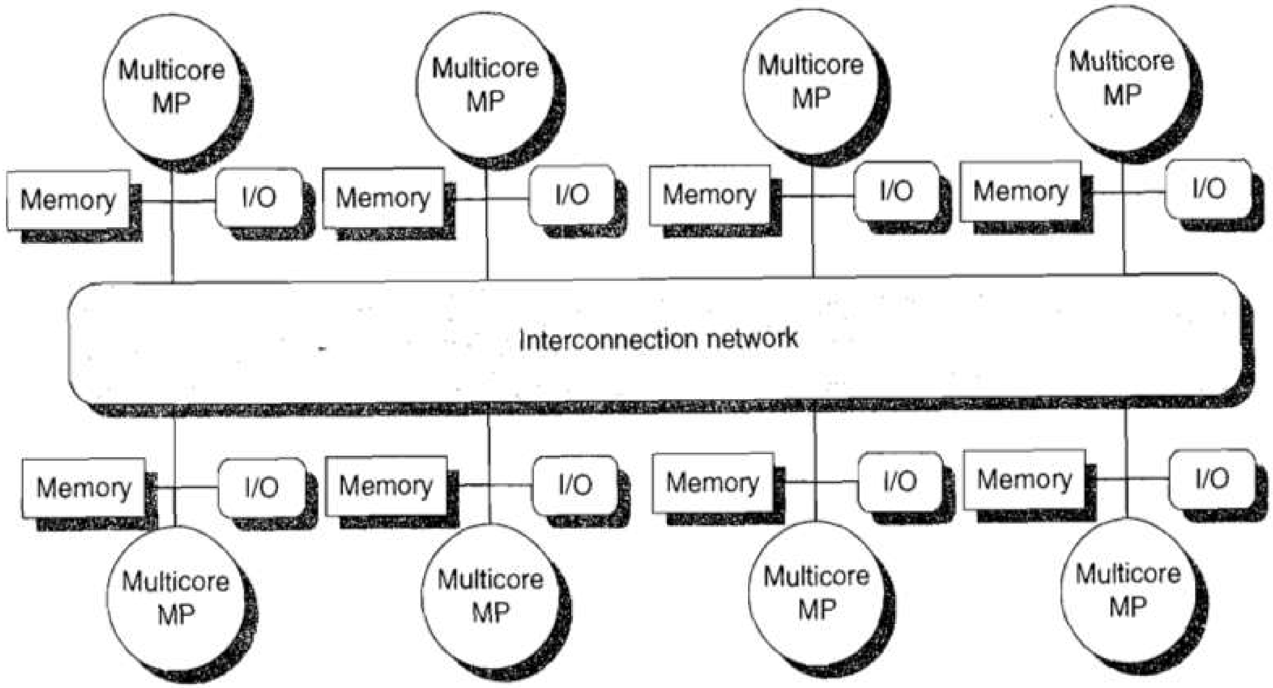


Figure 5.2 The basic architecture of a distributed-memory multiprocessor in 2011 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes. Each processor core shares the entire memory, although the access time to the lock memory attached to the core's chip will be much faster than the access time to remote memories.

Nodes and [N]UMA

The Linux kernel organizes physical RAM depending on whether we're on a [N]UMA machine ([Non] Uniform Memory Access). (Actually, the difference is minimal from the viewpoint of most generic mm code).

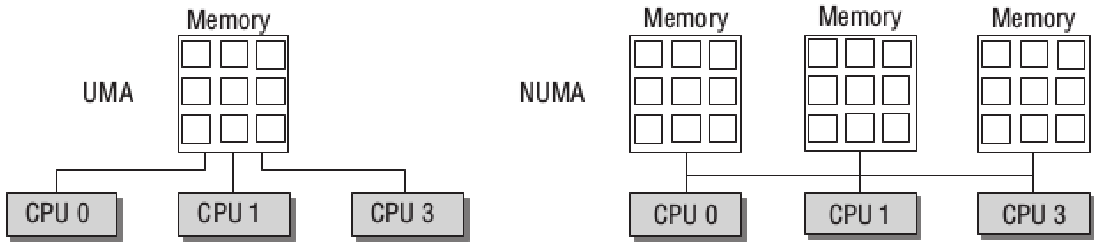


Figure 3-1: UMA and NUMA systems.

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

Source: Documentation/vm/numa
"…
Linux divides the system's hardware resources into multiple software abstractions called "nodes". Linux maps the nodes onto the physical cells of the hardware platform, abstracting away some of the details for some architectures. As with physical cells, software nodes may contain 0 or more CPUs, memory and/or IO buses. And, again, memory accesses to memory on "closer" nodes--nodes that map to closer cells--will generally experience faster access times and higher effective bandwidth than accesses to more remote cells.

<<
Quick way to check number of NUMA nodes (run on an x86_64 UMA box which shows up of course as pseudo-NUMA !) :

```
$ numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 7871 MB
node 0 free: 348 MB
node distances:
node   0
  0:  10
$
```

The **numactl** utility is useful: one can use it to define and control NUMA policy for processes or shared memory segments.
>>

…
For each node with memory, Linux constructs an independent memory management subsystem, complete with its own free page lists, in-use page lists, usage statistics and locks to mediate access. In addition, Linux constructs for each memory zone [one or more of DMA, DMA32, NORMAL, HIGH_MEMORY, MOVABLE], an ordered "zonelist". A zonelist specifies the zones/nodes to visit when a selected zone/node cannot satisfy the allocation request. This situation, when a zone has no available memory to satisfy a request, is called "overflow" or "fallback".
...”

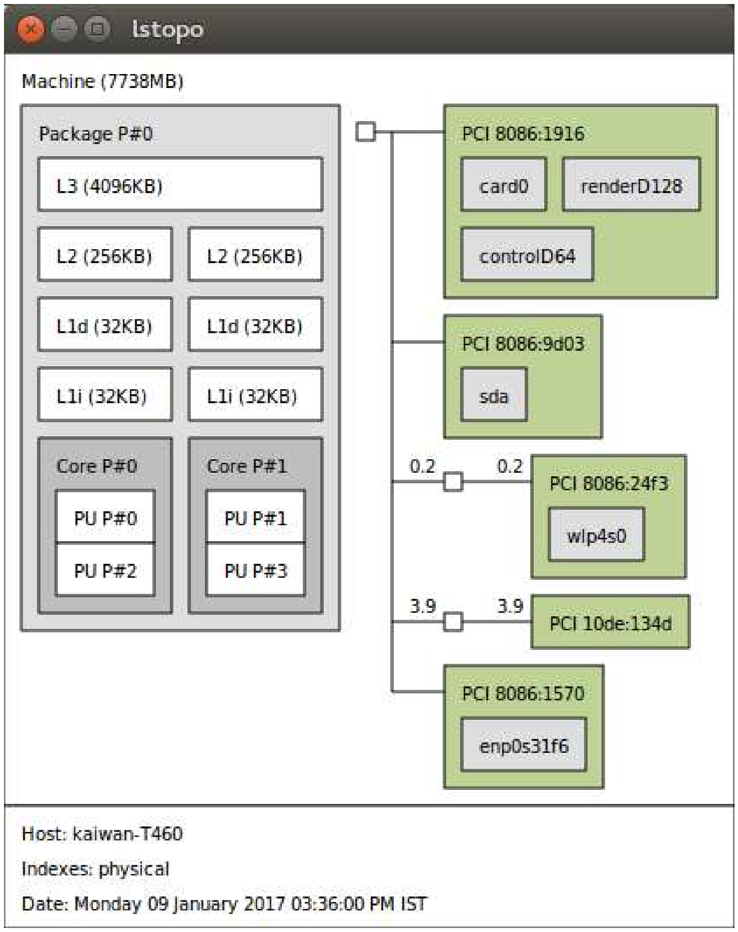
<<

- *See the “Red Hat Enterprise Linux 6 Performance Tuning Guide” section 4.1 “CPU Topology” for some details on NUMA and how it affects performance.*
  (Also FYI: “Red Hat Enterprise Linux 7 Performance Tuning Guide” PDF)

- *‘lscpu’ CLI tool (install the ‘hwloc’ package):*
  Output on a COTS (commercial off-the-shelf) Intel Core-i7 laptop (the Lenovo T460):

```
# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 78
Model name:            Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
Stepping:              3
CPU MHz:               579.617
CPU max MHz:           3100.0000
CPU min MHz:           400.0000
BogoMIPS:              5183.84
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              4096K
NUMA node0 CPU(s):     0-3
...
```

- *Check out the '**lstopo**' utiltity: it shows a graphical representation of CPU topology! Output on a COTS (commercial off-the-shelf) Intel Core-i7 laptop (the Lenovo T460):*



>>

[Source: Memory Access Patterns are Important by Martin Thompson]

...

Non-Uniform Memory Access (NUMA)

Systems now have memory controllers on the CPU socket. This move to on-socket memory controllers gave an ~50ns latency reduction over existing front side bus (FSB) and external [Northbridge] memory controllers. Systems with multiple sockets employ memory interconnects, [QPI] from Intel, which are used when one CPU wants to access memory managed by another CPU socket. The presence of these interconnects gives rise to the non-uniform nature of server memory access. In a 2-socket system memory may be local or 1 hop away. On a 8-socket system memory can be up to 3 hops away, where each hop adds 20ns latency in each direction.

...

>>

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

IBM

NUMA-Q server

HP Integrity Superdome-X server

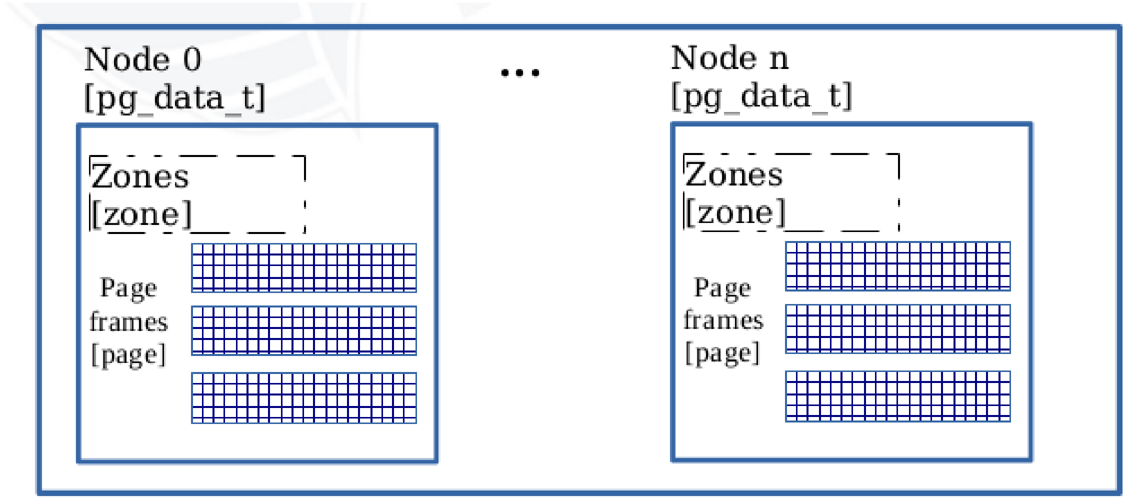
Examples of NUMA systems include Alpha-based Wildfire servers, NUMA-Q machines from IBM, HP Superdome servers, etc.

<< Source (below): "Professional Linux Kernel Architecture", W Mauerer, Wrox Press >>

On a NUMA system:

- 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* (seen below)
- Each node is further split into (at least one, upto four) *zones* (discussed below)
  - A zone is represented by the data structure *zone*
  - 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*

Physical RAM



[Optional / FYI]

Memory Models

We understand that, broadly, there are two types of machines that manage physical memory in different ways – UMA and NUMA systems.

A mix of both machine types with discontiguous memory is also possible. Such a mix would then represent a UMA system whose RAM is not contiguous but has large holes. Here it is often helpful to apply the principles of NUMA organization to make memory access simpler for the kernel.

In fact, the kernel distinguishes three configuration options — FLATMEM , DISCONTIGMEM , and SPARSEMEM .

SPARSEMEM and DISCONTIGMEM serve practically the same purpose, but in the view of developers, differ in the quality of their code — SPARSEMEM is regarded as more experimental and less stable but does feature performance optimizations.

Discontiguous memory is presumed to be more stable, but is not prepared for new features like memory hotplugging.

<<
In fact, the generic x86_64 “PC” architecture is exactly like this: it's UMA but large “holes” are possible in the physical memory space. The x86_64 default kernel configuration file (access via '`make menuconfig`') for Linux OS uses the SPARSEMEM option by default :

```
“Symbol: SPARSEMEM_MANUAL [=y]
   Type  : boolean
 Prompt: Sparse Memory
   Location:
    -> Processor type and features
      -> Memory model (<choice> [=y])
  Defined at mm/Kconfig:47
...
”
```

```
$ cat mm/Kconfig
config SELECT_MEMORY_MODEL
        def_bool y
        depends on ARCH_SELECT_MEMORY_MODEL

choice
         prompt "Memory model"
        depends on SELECT_MEMORY_MODEL
        default DISCONTIGMEM_MANUAL if ARCH_DISCONTIGMEM_DEFAULT
        default SPARSEMEM_MANUAL if ARCH_SPARSEMEM_DEFAULT
        default FLATMEM_MANUAL

...

config SPARSEMEM_MANUAL
        bool "Sparse Memory"
        depends on ARCH_SPARSEMEM_ENABLE
        help
          This will be the only option for some systems, including
          memory hotplug systems.  This is normal.

          For many other systems, this will be an alternative to
          "Discontiguous Memory".  This option provides some potential
          performance benefits, along with decreased code complexity,
          but it is newer, and more experimental.
...
```

>>

In the following sections, we restrict ourselves largely to FLATMEM because this memory organization type is used on most configurations and is also usually the kernel default. The fact that we do not discuss the other options is no great loss because all memory models make use of practically the same data structures.

Resources
LWN: sparsemem Memory Model
…
Sparsemem abstracts the use of discontiguous mem_maps[]. This kind of mem_map[] is needed by discontiguous memory machines (like in the old CONFIG_DISCONTIGMEM case) as well as memory hotplug systems.
Sparsemem replaces DISCONTIGMEM when enabled, and it is hoped that it can eventually become a complete replacement.

A significant advantage over DISCONTIGMEM is that it's completely separated from CONFIG_NUMA. When producing this patch, it became apparent in that NUMA and DISCONTIG are often confused.
…

Kernel ver 2.6.21
Generic Virtual Memmap suport for SPARSEMEM
"
…
However, if there is enough virtual space available and the arch already maps its 1-1 kernel space using TLBs (f.e. true of IA64 and x86_64) then this technique makes sparsemem lookups as efficient as CONFIG_FLATMEM.

Maybe this patch will allow us to make SPARSEMEM the default configuration that will work on UP, SMP and NUMA on most platforms?
Then we may hopefully be able to remove the various forms of support for FLATMEM, DISCONTIG etc etc.

Signed-off-by: Christoph Lameter <clameter@sgi.com>
… "

Emulating NUMA Nodes with Qemu

Qemu allows us to emulate NUMA nodes. From the man page:

```
…
  -cpu model
           Select CPU model ("-cpu help" for list and additional feature selection)
```

```
       -smp [cpus=]n[,cores=cores][,threads=threads][,sockets=sockets]
[,maxcpus=maxcpus]
           Simulate an SMP system with n CPUs. On the PC target, up to 255 CPUs are
supported. On Sparc32 target, Linux limits the number of usable CPUs to 4.  For the
PC target, the number of cores per socket, the number of threads per cores and the
total number of sockets can be specified. Missing values will be computed. If any on
the three values is given, the total number of CPUs n can be omitted. maxcpus
specifies the maximum number of hotpluggable CPUs.

       -numa node[,mem=size][,cpus=cpu[-cpu]][,nodeid=node]
       -numa node[,memdev=id][,cpus=cpu[-cpu]][,nodeid=node]
           Simulate a multi node NUMA system. If mem, memdev and cpus are omitted,
resources are split equally. Also, note that the -numa option doesn't allocate any of
the specified resources. That is, it just assigns existing resources to NUMA nodes.
This means that one still has to use the -m, -smp options to allocate RAM and VCPUs
respectively, and possibly -object to specify the memory backend for the memdev
suboption.

           mem and memdev are mutually exclusive.  Furthermore, if one node uses
memdev, all of them have to use it.
...
```

```
$ 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

```
root@kaiwan-ThinkPad-X220:~# uname -r
4.4.21
root@kaiwan-ThinkPad-X220:~# cat /etc/issue
Debian GNU/Linux 7 \n \l

root@kaiwan-ThinkPad-X220:~# free -h
             total       used       free     shared    buffers     cached
Mem:          996M        38M       958M         0B       1.0M       9.9M
-/+ buffers/cache:        27M       969M
Swap:           0B         0B         0B
root@kaiwan-ThinkPad-X220:~# cat /proc/buddyinfo
Node 0, zone      DMA      9      6      6      7      4      3      3      2
  2      3      1
Node 0, zone    DMA32    216    136     99     55     31     21     16      4
  5      0    110
Node 1, zone    DMA32    195    377    282    120     50     24     11      4
  0      2    114
root@kaiwan-ThinkPad-X220:~# _
```

Notice above in the output of /proc/buddyinfo, the two nodes – Node 0 and Node 1!

… and a capture of Qemu's monitor:

```
QEMU 2.6.1 monitor - type 'help' for more information
(qemu)
(qemu) info cpu
cpus      cpustats
(qemu) info cpus
* CPU #0: pc=0xffffffff8100bf96 (halted) thread_id=6621
```

```
  CPU #1: pc=0xffffffff8100bf96 (halted) thread_id=6622
  CPU #2: pc=0xffffffff8100bf96 (halted) thread_id=6623
  CPU #3: pc=0xffffffff8100bf96 (halted) thread_id=6624
(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
(qemu)
```

Emulation of NUMA via the "fake-NUMA" kernel config

Real NUMA systems will set the configuration option CONFIG_NUMA, and the memory management codes will differ between the two variants. Since the flat memory model will not make sense on NUMA machines, only discontiguous and sparse memory will be available. Notice that the configuration option NUMA_EMU allows AMD64 systems with a flat memory to enjoy the full complexities of NUMA systems by splitting the memory into fake NUMA zones. This can be useful for development when no real NUMA machine is available — for some reason, these tend to be rather costly.

Ref: http://linux-hacks.blogspot.in/2009/07/fake-numa-nodes-in-linux.html

---

(Here we) focus on the UMA case, and does not consider CONFIG_NUMA. This does not mean that the NUMA data structures can be completely neglected. Since UMA systems can choose the configuration option CONFIG_DISCONTIGMEM if their address space contains large holes, then more than one memory node can also be available on systems that do not employ NUMA techniques otherwise.

[Aside / possibly useful:
Also see: *https://www.kernel.org/doc/Documentation/ABI/stable/sysfs-devices-node*

3.8 Linux: Automatic NUMA balancing

A lot of modern machines are "non uniform memory access" (NUMA) architectures: they have per-processor memory controllers, and accessing the memory in the local processor is faster than accessing the memory of other processors, so the placement of memory in the same node where processes will reference it is critical for performance. This is specially true in huge boxes with docens or hundreds of processors.

The Linux NUMA implementation had some deficiencies. This release includes a new NUMA foundation which will allow to build smarter NUMA policies in the next releases. For more details, see the LWN article:

Recommended LWN article: NUMA in a hurry

]

UMA

UMA machines can be UP or SMP; all CPUs access RAM equally fast. In practice, in order to keep the mm code as generic as possible, an UMA system's memory organization is identical to NUMA except that we use just a *single node*. Everything else remains the same!

In fact, on UMA, a single NUMA node manages the entire physical memory and other parts of the MM system are led to believe that they are working on a pseudo-NUMA machine.

File : [3.10.24] include/linux/mm.h:

```
...
/*
 * The pg_data_t structure is used in machines with CONFIG_DISCONTIGMEM
 * (mostly NUMA machines?) to denote a higher-level memory zone than the
 * zone denotes.
 *
 * On NUMA machines, each NUMA node would have a pg_data_t to describe
 * it's memory layout.
 *
 * Memory statistics and page replacement data structures are maintained on
 * a per-zone basis.
 */
struct bootmem_data;
typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];
        struct zonelist node_zonelists[MAX_ZONELISTS];
        int nr_zones;
#ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM */
    struct page *node_mem_map;

      ...

      unsigned long node_start_pfn;
      unsigned long node_present_pages; /* total number of physical pages */
      unsigned long node_spanned_pages; /* total size of physical page
                                           range, including holes */
      int node_id;
      nodemask_t reclaim_nodes;   /* Nodes allowed to reclaim from */
      wait_queue_head_t kswapd_wait;
      wait_queue_head_t pfmemalloc_wait;
      struct task_struct *kswapd; /* Protected by lock_memory_hotplug() */ << for
swapping >>
      int kswapd_max_order;
      enum zone_type classzone_idx;
...
} pg_data_t;
```

FYI / OPTIONAL

...

Architectures are responsible for setting up a CPU ID to NUMA memory node mapping.

File : [kernel ver 3.10.24] arch/arm/include/asm/setup.h *(ARM-specific, as an example)*

```
...
/*
 * Memory map description
 */
#define NR_BANKS    CONFIG_ARM_NR_BANKS

struct membank {
    phys_addr_t start;
    phys_addr_t size;
    unsigned int highmem;
};

struct meminfo {
    int nr_banks;
    struct membank bank[NR_BANKS];
};

extern struct meminfo meminfo;

#define for_each_bank(iter,mi)              \
    for (iter = 0; iter < (mi)->nr_banks; iter++)
...
```

There also exists Linux-specific NUMA API support – via system calls and libnuma; please see numa(7) man page for details.

Also:
/proc/[number]/numa_maps (since Linux 2.6.14)
This file displays information about a process's NUMA memory policy and allocation.

Zones

(Based on notes from "Professional Linux Kernel Architecture", Maurerer, Wrox Press)

Each node is split into zones as further subdivisions of memory. For example, there are restrictions as to the memory area that can be used for DMA operations (with ISA devices); only the first 16 MiB are suitable. There is also a highmem area that cannot be mapped directly (by the kernel). Between these is the ''normal" memory area for universal use. A node therefore comprises up to three zones.

<linux/mmzone.h>

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
    /*
     * ZONE_DMA is used when there are devices that are not able
     * to do DMA to all of addressable memory (ZONE_NORMAL). Then we
     * carve out the portion of memory that is needed for these devices.
     * The range is arch specific.
     *
     * Some examples
     *
     * Architecture     Limit
     * ---------------------------
     * parisc, ia64, sparc  <4G
     * s390         <2G
     * arm          Various
     * alpha        Unlimited or 0-16MB.
     *
     * i386, x86_64 and multiple other arches
     *          <16M.
     */
    ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
    /*
     * x86_64 needs two ZONE_DMAs because it supports devices that are
     * only able to do DMA to the lower 16M but also 32 bit devices that
     * can only do DMA areas below 4G.
     */
    ZONE_DMA32,
#endif
    /*
     * Normal addressable memory is in ZONE_NORMAL. DMA operations can be
     * performed on pages in ZONE_NORMAL if the DMA devices support
     * transfers to all addressable memory.
     */
    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
    /*
    * A memory area that is only addressable by the kernel through
     * mapping portions into its own address space. This is for example
     * used by i386 to allow the kernel to address the memory beyond
```

```
     * 900MB. The kernel will set up special mappings (page
     * table entries on i386) for each page that the kernel needs to
     * access.
     */
    ZONE_HIGHMEM,
#endif
    ZONE_MOVABLE,
    __MAX_NR_ZONES
};
```

❑ ZONE_DMA for DMA-suitable memory. The size of this region depends on the processor type. On IA-32 machines, the limit is the classical 16 MiB boundary imposed by ancient ISA devices. But also, more modern machines can be affected by this.

From 2.6.21, there is a kernel configuration directive regarding memory allocation specifically from ZONE_DMA:

```
[*] DMA memory allocation support
Help:
CONFIG_ZONE_DMA:
 DMA memory allocation support allows devices with less than 32-bit
 addressing to allocate within the first 16MB of address space.

 Disable if no such devices will be used.
 If unsure, say Y.
```

❑ ZONE_DMA32 for DMA-suitable memory in a 32-bit addressable area. Obviously, there is only a difference between the two DMA alternatives on 64-bit systems. On 32-bit machines, this zone is empty; that is, its size is 0 MiB. On Alphas and AMD64 systems, for instance, this zone ranges from 0 to 4 GiB.

❑ ZONE_NORMAL for normal memory mapped directly in the kernel segment. This is the only zone guaranteed to be possible present on all architectures. It is, however, not guaranteed that the zone must be equipped with memory. If, for instance, an AMD64 system has 2 GiB of RAM, then all of it will belong to ZONE_DMA32, and ZONE_NORMAL will be empty.

❑ ZONE_HIGHMEM for physical memory that extends beyond the kernel segment.

Depending on the compile-time configuration, some zones need not be considered. 64-bit systems, for instance, do not require a high memory zone, and the DMA32 zone is only required on 64-bit systems that also support 32-bit peripheral devices that can only access memory up to 4 GiB.

Each zone is associated with an array in which the physical memory pages belonging to the zone — known as page frames in the kernel — are organized. An instance of struct page with the required management data is allocated for each page frame.

The nodes are kept on a singly linked list so that the kernel can traverse them.

For performance reasons, the kernel always attempts to perform the memory allocations of a process on the NUMA node associated with the CPU on which it is currently running. However, this is not always possible — for example, the node may already be full. For such situations, each node provides a fallback list (with the help of struct zonelist). The list contains other nodes (and associated zones) that can be used as alternatives for memory allocation. The further back an entry is on the list, the less suitable it is.

What's the situation on UMA systems? Here, there is just a single node — no others.

…

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

```
struct zone {
       /* Fields commonly accessed by the page allocator */
       /* zone watermarks, access with *_wmark_pages(zone) macros */
      unsigned long watermark[NR_WMARK];
--snip--
      unsigned long       lowmem_reserve[MAX_NR_ZONES];
      struct per_cpu_pageset __percpu *pageset;
--snip--
        spinlock_t              lock;
--snip--
       struct free_area       free_area[MAX_ORDER];
--snip--
        ZONE_PADDING(_pad1_)

       /* Fields commonly accessed by the page reclaim scanner */
        spinlock_t              lru_lock;
<<
 File : [3.10.24] : include/linux/mmzone.h
  /*
   * zone->lock and zone->lru_lock are two of the hottest locks in the kernel.
   * So add a wild amount of padding here to ensure that they fall into separate
   * cachelines.  There are very few zone structures in the machine, so space
   * consumption is not a concern here.
   */
>>
        struct list_head        active_list;
        struct list_head        inactive_list;
        unsigned long           nr_scan_active;
        unsigned long           nr_scan_inactive;
        unsigned long           pages_scanned;     /* since last reclaim */
        unsigned long           flags;          /* zone flags, see below */
--snip--
        unsigned int inactive_ratio;


    ZONE_PADDING(_pad2_)
    /* Rarely used or read-mostly fields */
--snip--
    unsigned long       spanned_pages;
    unsigned long       present_pages;
    unsigned long       managed_pages;
```

```
    /*
     * rarely used fields:
     */
    const char      *name;
} ____cacheline_internodealigned_in_smp;
```

The *lowmem_reserve* array specifies several pages for each memory zone that are reserved for critical allocations that must not fail under any circumstances. Each zone contributes according to its importance. The algorithm to calculate the individual contributions is discussed in Section 3.2.2.

pageset is an array to implement per-CPU hot-n-cold page lists. The kernel uses these lists to store fresh pages that can be used to satisfy implementations. However, they are distinguished by their cache status: Pages that are most likely still cache-hot and can therefore be quickly accessed are separated from cache-cold pages.

SIDEBAR
Structure Padding and Cache Lines
The striking aspect of this structure is that it is divided into several sections separated by ZONE_PADDING. This is because zone structures are very frequently accessed. On multiprocessor systems, it commonly occurs that different CPUs try to access structure elements at the same time. Locks are therefore used to prevent them interfering with each, and giving rise to errors and inconsistencies. The two spinlocks of the structure — zone->lock and zone->lru_lock — are often acquired because the kernel very frequently accesses the structure << known as *hotspots* >>. Data are processed faster if they are held in a cache of the CPU. Caches are divided into lines, and each line is responsible for various memory areas. The kernel invokes the ZONE_PADDING macro to generate "padding" that is added to the structure to ensure that each lock is in its own cache line. << [ULVMM] … inclusion of padding of zeros in the struct. Development of the 2.6 VM recognised that some spinlocks are very heavily contended and are frequently acquired. As it is known that some locks are almost always acquired in pairs, an effort should be made to ensure they use different cache lines which is a common cache programming trick [Sea00]. These padding in the struct zone are marked with the ZONE_PADDING() macro and are used to ensure the zone→lock, zone→lru_lock and zone→pageset fields use different cache lines. >> The compiler keyword __cacheline_maxaligned_in_smp is also used to achieve optimal cache alignment.

The last two sections of the structure are also separated from each other by padding. As neither includes a lock, the primary aim is to keep the data in a cache line for quick access and thus to dispense with the need for loading the data from RAM memory, which is a slow process. The increase in size due to the padding structures is negligible, particularly as there are relatively few instances of zone structures in kernel memory.
Source- Professional Linux Kernel Architecture, Maurerer, Wrox.

The opposite of structure padding is structure packing- gcc can achieve this by using the "__packed" keyword. Eg.:
`arch/x86/kvm/vmx.c`
`...`
`* This structure is packed to ensure that its layout is identical across`
`* machines (necessary for live migration).`
`struct __packed vmcs12 { … }; << fyi, this struct is to do with nesting a kvm guest within a guest! >>`

See the Appendices at the end of this topic for details on CPU TLB and Cache management.
Useful Resource: "What Every Programmer should know about Memory", Ulrich Drepper.

<<
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.
>>

Zone structure (contd.)

The structure is big, but there are only three zones in the system and, thus, only three of these structures. Let's look at the more important fields. The lock field is a spin lock that protects the structure from concurrent access. Note that it protects just the structure, and not all the pages that reside in the zone. A specific lock does not protect individual pages, although parts of the kernel may lock the data that happens to reside in said pages.

The free_pages field is the number of free pages in this zone (older kernel ver). The kernel tries to keep at least pages_min pages free (through swapping), if possible.

The name field is, unsurprisingly, a NULL-terminated string representing the name of this zone. The kernel initializes this value during boot in *mm/page_alloc.c* and the three zones are given the names "DMA," "Normal," and "HighMem."

Dynamic Memory Allocation

[1] The goal of memory management is to provide a method by which memory can be dynamically shared amongst a variety of users for a variety of purposes. The memory management method should do both of the following:

- Minimize the amount of time required to manage the memory
- Maximize the available memory for general usage (minimize management overhead)

Memory management is ultimately a zero-sum game of tradeoffs. You can develop an algorithm that uses little memory for management but takes more time to manage the available memory. You can also develop an algorithm that efficiently manages memory but uses a bit more memory. In the end, the requirements for the particular application drive the balance of the tradeoffs.

Heap-Based Allocation Strategy

Early memory managers used a heap-based allocation strategy. In this method, a large block of memory (called the heap) is used to provide memory for user-defined purposes. When users need a block of memory, they make a request for a given size. The heap manager looks at the available memory (using a particular algorithm) and returns the block. Some of the algorithms used in this search are the *first-fit* (the first block encountered in the heap that satisfies the request), and the *best-fit* (the best block in the heap that satisfies the request). When the users are finished with the memory, they return it to the heap.

The fundamental problem with this heap-based allocation strategy is fragmentation. As blocks of memory are allocated, they are returned in different orders and at different times. This tends to leave holes in the heap requiring more time to efficiently manage the free memory. This algorithm tends to be memory efficient (allocating what's necessary) but requires more time to manage the heap.

Buddy-System Allocation Strategy

Another approach, called buddy memory allocation, is a faster memory technique that divides memory into power-of-2 partitions and attempts to allocate memory requests using a best-fit approach. When memory is freed by the user, the buddy block is checked to see if any of its contiguous neighbors have also been freed. If so, the blocks are combined to minimize fragmentation. This algorithm tends to be a bit more time efficient but can waste memory due to the best-fit approach.

[1] Above page extracted from the article “Anatomy of the Linux Slab Allocator” by M. Tim Jones, published by IBM DeveloperWorks [http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/].

The Buddy System

Some notes from here.

Within each zone, the *buddy system* is used to **manage physical page frames**.

The buddy system solves to a large extent the typical problem of *external fragmentation of memory* - very frequent requests & releases of chunks of contiguous page frames of different sizes lead to a situation where free pages are scattered inside blocks of allocated page frames. Due to this, even having enough individual free page frames, it may be impossible to allocate a single large contiguous block of page frames.

Most page allocation and freeing is done when allocating and de-allocating pages for process virtual address space. Since such allocations usually occur during page-fault processing, the majority of page allocations are for a single page. However, some kernel entities, particularly device drivers, have special needs with respect to physical memory. One might, for example, need to allocate 4 contiguous pages of DMA-capable physical RAM. **The buddy system allows such allocations to be done quickly and efficiently**.

Essentially, the buddy system treats physical RAM as a collection of 2^n page-sized blocks aligned on 2^n-page boundaries, and **merges adjacent free blocks into single higher-order blocks**. Each 2^n-page block is the "buddy" of the other half of the 2^{n+1}-page block containing it.

The allocator keeps lists of all the free one-page blocks, all the free two-page, two-page-aligned blocks, all the free four-page, four-page-aligned blocks, etc.

All free page frames are grouped into 11 lists of blocks (called *freelists;* MAX_ORDER=11 defined in <*linux/mmzone.h*>) that contain groups of (powers of 2 page frames - called the *order* of the block/freelist) - 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 contiguous page frames respectively (corresponding to order 0 to 10 : as 2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9 and 2^{10}).

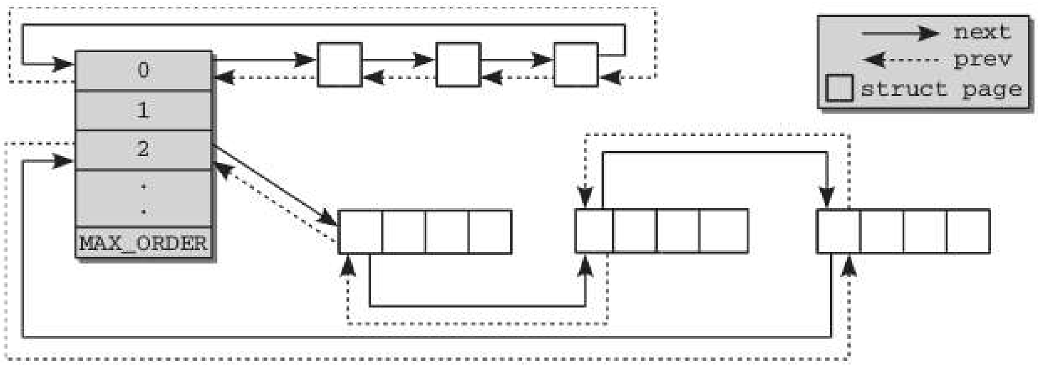


Figure 3-22: Linking blocks in the buddy system.

Above diagram source: Professional Linux Kernel Architecture by Wolfgang Mauerer, Wrox Press.

```
# cat /proc/buddyinfo
Node 0, zone DMA        85    56    41    11    20  16     1     1     0     1     0
Node 0, zone Normal   1378  1694  1318   226    87   5     5    47    20     1     0
Node 0, zone HighMem 12510   409     2     0     0   0     0     0     0     0     0
#
```

Output below on an x86_64 VM with 1 GB RAM

```
# cat /proc/buddyinfo
Node 0, zone      DMA     64    113     67      7      8      2      2      2      2      0      0
Node 0, zone    DMA32    977   7029   4735    924    194     57     28     13      8      0     13
#
```

<<
An article on examining /proc/buddyinfo and a Python script to make the output even more human readable.
>>
<<
From *include/linux/mmzone.h*

```
--snip--

 /* Free memory management - zoned buddy allocator.  */
 #ifndef CONFIG_FORCE_MAX_ZONEORDER
 #define MAX_ORDER 11
 #else
 #define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER
 #endif
 #define MAX_ORDER_NR_PAGES (1 << (MAX_ORDER - 1))  << = 1024 >>
...
>>
```

<<
Source: Professional Linux Kernel Architecture by Wolfgang Mauerer, Wrox Press.

The typical value of this constant is 11, which means that the maximum number of pages that can be requested in a single allocation is 2^11 = 2, 048 (page frames). However, this value can be changed manually if the FORCE_MAX_ZONEORDER configuration option is set by the architecture-specific code.

For example, the gigantic address spaces on IA-64 systems allow for working with MAX_ORDER = 18 [2^18 = 262,144 page frames => largest size block is 1 GB!], whereas ARM or v850 systems use smaller values such as 8 or 9. This, however, is not necessarily caused by little memory supported by the machine, but can also be because of memory alignment requirements.
…

Memory management based on the buddy system is concentrated on a single memory zone of a node, for instance, the DMA or high-memory zone.

However, the buddy systems of all zones and nodes are linked via the allocation fallback list. Figure 3-23 illustrates this relationship.

When a request for memory cannot be satisfied in the preferred zone or node, first another zone in the same node, and then another node is picked to fulfill the request.

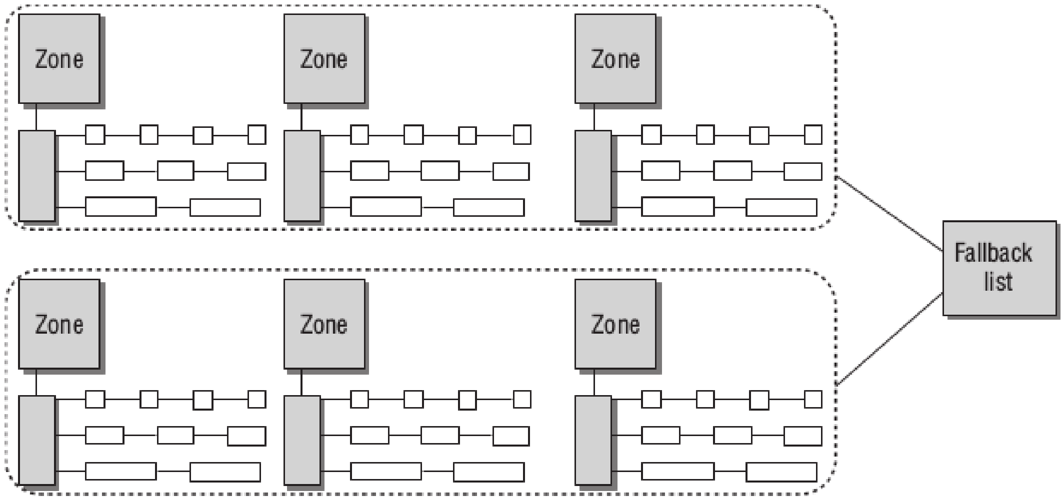


Figure 3-23: Relationship between buddy system and memory zones/nodes.

>>

To allocate a block of a given order, we check the freelist of the specified order and all higher orders. If a block is found at the specified order, it is allocated immediately. If a block of higher order must be used, then we **divide the larger block** into two $2^{order-1}$ blocks, add the lower half to the appropriate freelist, and allocate the memory from the upper half, executing this step recursively if necessary.

When freeing memory, we check whether the block being freed has a free buddy block; if so, we combine the two blocks into a single free block by removing the adjacent block from its freelist and then freeing the larger block; again, this process is performed recursively if necessary.

<<

File : include/linux/mmzone.h

```
...
 83 struct free_area {
 84     struct list_head    free_list[MIGRATE_TYPES]; << MIGRATE_TYPES: seen
                                                              later >>
 85     unsigned long       nr_free;  << # of free 2^n page blocks >>
 86 };

--snip--

File : [3.10.24]: include/linux/mmzone.h

struct zone {
        /* Fields commonly accessed by the page allocator */
        /* zone watermarks, access with *_wmark_pages(zone) macros */
       unsigned long watermark[NR_WMARK];
--snip--
        struct free_area        free_area[MAX_ORDER];
--snip--
         ZONE_PADDING(_pad1_)
```

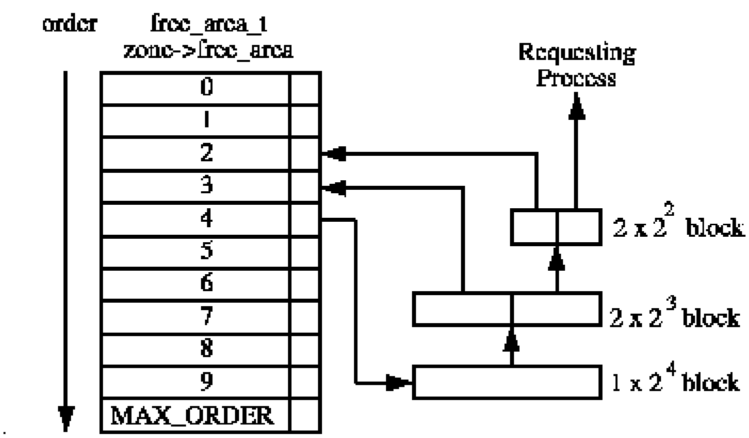
```
    /*
     * rarely used fields:
     */
    const char      *name;
} ____cacheline_internodealigned_in_smp;
...
>>
```

<<

[*Source - ULVMM*](#)

...

Allocations are always for a specified order, 0 in the case where a single page is required. If a free block cannot be found of the requested order, a higher order block is split into two buddies. One is allocated and the other is placed on the free list for the lower order. Figure 6.2 shows where a 2^4 block is split and how the buddies are added to the free lists until a block for the process is available.



<< Note: MAX_ORDER is 11 (thus 0-10) from 2.6 onwards >>

Figure 6.2: Allocating physical pages

When the block is later freed, the buddy will be checked. If both are free, they are merged to form a higher order block and placed on the higher free list where its buddy is checked and so on. If the buddy is not free, the freed block is added to the free list at the current order. [During these list manipulations, interrupts have to be disabled to prevent an interrupt handler manipulating the lists while a process has them in an inconsistent state. This is achieved by using an interrupt safe spinlock.]

The second decision to make is which memory node or *pg_data_t* to use. Linux uses a node-local allocation policy which aims to use the memory bank associated with the CPU running the page allocating process. Here, the *function_alloc_pages()* is what is important as this function is different depending on whether the kernel is built for a UMA (function in mm/page_alloc.c) or NUMA (function in mm/numa.c) machine.

<< Note: In 2.4, there was specific code dedicated to selecting the correct node to allocate from based on the unning CPU but *2.6 removes this distinction between NUMA and UMA architectures*. … (Also), architectures are responsible for setting up a CPU ID to NUMA memory node mapping. >>

Regardless of which API is used, *__alloc_pages_nodemask()* in *mm/page_alloc.c* is the heart of the allocator.

<< *Update:*

```
mm/page_alloc.c
/*
 * This is the 'heart' of the zoned buddy allocator.
 */
struct page *
 __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
             struct zonelist *zonelist, nodemask_t *nodemask)
...
>>
```

This function, which is never called directly, examines the selected zone and checks if it is suitable to allocate from based on the number of available pages. If the zone is not suitable, the allocator may fall back to other zones.

The order of zones to fall back on are decided at boot time by the function build_zonelists() but generally ZONE_HIGHMEM will fall back to ZONE_NORMAL and that in turn will fall back to ZONE_DMA. If number of free pages reaches the pages_low watermark, it will wake **kswapd** to begin freeing up pages from zones and if memory is extremely tight, the caller will do the work of **kswapd** itself.

...

>>

<<
Src: How a simple Linux kernel memory corruption bug can lead to complete system compromise, Jann Horn, Google Project Zero, Oct 2021

"… When the page is given to the page allocator, we benefit from the page being order-0 (4 KiB, native page size): For order-0 pages, the page allocator has special freelists, one per CPU+zone+migratetype combination. Pages on these freelists are not normally accessed from other CPUs, and they don't immediately get combined with adjacent free pages to form higher-order free pages. ..."

>>

<<
Complexities

The buddy system alllocator, indeed, the entire memory management code, has in reality much complexity to deal with. Many issues exist:

- Performance
  - fragmentation
  - as few “lock” paths as possible
  - minimize free page searching by keeping similar (migration) types of pages together
  - NUMA considerations
- Large physically contiguous memory regions allocation
- etc..

These have given rise to various solutions, many of which are now deeply merged into the fabric of Linux MM; these include the notions of (and corresponding technology) :
- pagesets
- page migration types
- CMA .

For details, please see “Appendix I - *(Some) VM Complexities*"
>>

Additional Note:
One way of seeing memory usage statistics is via the “Magic SysRq” facility: Alt-SysRq-M. For details, see *Documentation/sysrq.txt*

1.
```
# echo m > /proc/sysrq-trigger
# dmesg
[...]
```

<<
Migration Types

In the output of 'echo m > /proc/sysrq-trigger' , what do the (UEM), etc flags mean?

Eg.:
…
Normal: 1015*4kB (UEM) 672*8kB (UEM) 670*16kB (UEM) 565*32kB (UEM) 394*64kB (UEM) …

Short answer: from 2.6.24 onward, the buddy system freelists are organized not only per NUMA node : Zone : Order 'n'
but within each zone we have further organization by page mobility group.

This is done to further avoid fragmenting memory; part of the Linux kernel's “anti-fragmentation” approach!

```
static void show_migration_types(unsigned char type)
{

static const char types[MIGRATE_TYPES] = {
[MIGRATE_UNMOVABLE] = 'U',
[MIGRATE_RECLAIMABLE] = 'E',
[MIGRATE_MOVABLE] = 'M',
[MIGRATE_RESERVE] = 'R',
#ifdef CONFIG_CMA
[MIGRATE_CMA] = 'C',
#endif
#ifdef CONFIG_MEMORY_ISOLATION
[MIGRATE_ISOLATE] = 'I',
#endif

};
…
```

So, prior to 2.6.24, the buddy-system can be visualized as:

```
                        :: order 0 list
                        :: order 1 list
Node R :: Zone          ::   --snip--
                        :: order n-2 list
                        :: order n-1 list ; where n = MAX_ORDER (11)
```

Now, from 2.6.24 onwards, for a system with N (NUMA) nodes and page mobility grouping:

```
                                      :: order 0 list
                                      :: order 1 list
Node R :: Zone :: Migration Type ::   --snip--
                                      :: order n-2 list
                                      :: order n-1 list ; where n = MAX_ORDER
                                                   (11 on ARM,x86)
```

```
 Node       Zone          Migration Type

Node 0 : Zone DMA      : MIGRATE_UNMOVABLE     :: order 0, 1, 2, … (n-1) lists
Node 0 : Zone DMA      : MIGRATE_RECLAIMABLE   :: order 0, 1, 2, … (n-1) lists
Node 0 : Zone DMA      : MIGRATE_MOVABLE
Node 0 : Zone DMA      : MIGRATE_RESERVED      ...
Node 0 : Zone DMA      : [MIGRATE_CMA]
Node 0 : Zone DMA      : [MIGRATE_ISOLATE]     :: order 0, 1, 2, … (n-1) lists

Node 0 : Zone NORMAL   : MIGRATE_UNMOVABLE     :: order 0, 1, 2, … (n-1) lists
Node 0 : Zone NORMAL   : MIGRATE_RECLAIMABLE   :: order 0, 1, 2, … (n-1) lists
```

```
Node 0 : Zone NORMAL   : MIGRATE_MOVABLE
Node 0 : Zone NORMAL   : MIGRATE_RESERVED      ...
Node 0 : Zone NORMAL   : [MIGRATE_CMA]
Node 0 : Zone NORMAL   : [MIGRATE_ISOLATE]     :: order 0, 1, 2, … (n-1) lists

Node 0 : Zone HIGHMEM  : MIGRATE_UNMOVABLE     :: order 0, 1, 2, … (n-1) lists
Node 0 : Zone HIGHMEM  : MIGRATE_RECLAIMABLE   :: order 0, 1, 2, … (n-1) lists
Node 0 : Zone HIGHMEM  : MIGRATE_MOVABLE
Node 0 : Zone HIGHMEM  : MIGRATE_RESERVED      ...
Node 0 : Zone HIGHMEM  : [MIGRATE_CMA]
Node 0 : Zone HIGHMEM  : [MIGRATE_ISOLATE]     :: order 0, 1, 2, … (n-1) lists

...
...

Node N-1 : Zone DMA      : MIGRATE_UNMOVABLE   :: order 0, 1, 2, … (n-1) lists
Node N-1 : Zone DMA      : MIGRATE_RECLAIMABLE :: order 0, 1, 2, … (n-1) lists
Node N-1 : Zone DMA      : MIGRATE_MOVABLE
Node N-1 : Zone DMA      : MIGRATE_RESERVED    ...
Node N-1 : Zone DMA      : [MIGRATE_CMA]
Node N-1 : Zone DMA      : [MIGRATE_ISOLATE]   :: order 0, 1, 2, … (n-1) lists

Node N-1 : Zone NORMAL   : MIGRATE_UNMOVABLE   :: order 0, 1, 2, … (n-1) lists
Node N-1 : Zone NORMAL   : MIGRATE_RECLAIMABLE :: order 0, 1, 2, … (n-1) lists
Node N-1 : Zone NORMAL   : MIGRATE_MOVABLE
Node N-1 : Zone NORMAL   : MIGRATE_RESERVED    ...
Node N-1 : Zone NORMAL   : [MIGRATE_CMA]
Node N-1 : Zone NORMAL   : [MIGRATE_ISOLATE]   :: order 0, 1, 2, … (n-1) lists

Node N-1 : Zone HIGHMEM  : MIGRATE_UNMOVABLE   :: order 0, 1, 2, … (n-1) lists
Node N-1 : Zone HIGHMEM  : MIGRATE_RECLAIMABLE :: order 0, 1, 2, … (n-1) lists
Node N-1 : Zone HIGHMEM  : MIGRATE_MOVABLE
Node N-1 : Zone HIGHMEM  : MIGRATE_RESERVED    ...
Node N-1 : Zone HIGHMEM  : [MIGRATE_CMA]
Node N-1 : Zone HIGHMEM  : [MIGRATE_ISOLATE]   :: order 0, 1, 2, … (n-1) lists

.
```

Take a look at the output of /proc/pagetypeinfo .

>>

SIDEBAR :: sysrq-m

The output of the above SysRq 'm' is very verbose and interesting to interpret!
Details: please refer to the Appendices associated with this module:

Linux VM | ***Appendix D :: Magic SysRq 'm' – the show_mem() Functionality***

>>

On 2.6 and onwards kernels:
For buddy system information, see */proc/buddyinfo* . For example:

`$ free -m`

```
             total       used       free     shared    buffers     cached
Mem:          2011       1402        609          0        110        600
-/+ buffers/cache:        690       1320
Swap:          996          6        989
$ cat /proc/buddyinfo
Node 0, zone  DMA         3      3    5    4    4    4    6    2    2    1    0
Node 0, zone  Normal  23304  11433 5713 2450 569 109    5    0    0    0    1
Node 0, zone  HighMem 14916   7615 3113  727 155  18    2    1    0    0    0
$
```

Some notes on VM and buddy system initialization can be seen here.

<< Source: ULVMM >>

Avoiding Fragmentation

One important problem that must be addressed with any allocator is the problem of internal and external fragmentation. External fragmentation is the inability to service a request because the available memory exists only in small blocks.

Internal fragmentation is defined as the wasted space where a large block had to be assigned to service a small request. In Linux, external fragmentation is not a serious problem as large requests for contiguous pages are rare and usually vmalloc() is sufficient to service the request. The lists of free blocks ensure that large blocks do not have to be split unnecessarily.

Internal fragmentation is the single most serious failing of the binary buddy system. While fragmentation is expected to be in the region of 28% it has been shown that it can be in the region of 60%, in comparison to just 1% with the first fit allocator[JW98]. It has also been shown that using variations of the buddy system will not help the situation significantly [PN77]. **To address this problem, Linux uses a slab allocator** [Bon94] to carve up pages into small blocks of memory for allocation. With this combination of allocators, the kernel can ensure that the amount of memory wasted due to internal fragmentation is kept to a minimum.

<<
IMP Note- from ver 2.6.27 Linux has a low-level API that mitigates the buddy system wastage factor: alloc_pages_exact(). Seen later!
>>

[FYI: The Buddy System on Wikipedia]

The Slab Cache

The slab allocator used in Linux is based on an algorithm first introduced by Jeff Bonwick for the SunOS operating system.

Jeff's allocator revolves around object caching. Within a kernel, a considerable amount of memory is allocated for a finite set of objects such as file descriptors and other common structures. Jeff found that the amount of time required to initialize a regular object in the kernel exceeded the amount of time required to allocate and deallocate it.

His conclusion was that instead of freeing the memory back to a global pool, he would have the memory remain initialized for its intended purpose. For example, if memory is being allocated for a mutex, the mutex initialization function (mutex_init) need only be performed once when the memory is first allocated for the mutex. Subsequent allocations of the memory need not perform the initialization because it's already in the desired state from the previous deallocation and call to the deconstructor.

The Linux slab allocator uses these ideas and others to build a memory allocator that is efficient in both space and time.

Also, using the buddy system described above, we can see that the minimum page "order" is 0 => 2^0 = 1 page *minimally* will be allocated using this system.
So how does the system allocate fragments of a page? The Linux solution is to layer the buddy system under a general-purpose allocator called the **Slab Allocator** (essentially borrowed from the Solaris MM system). This enables the allocation of memory less than a page (or of some general size) via the slab cache.

The slab cache services memory requests within kernel-space.

<< [Optional] Course Text Ref ::
"Linux Kernel Development" by Robert M Love 2nd Ed.
Ch 11 "Memory Management" section "Slab Layer" page 194.
>>

Source

The slab allocator has three principle aims:

- The allocation of small blocks of memory to help eliminate internal fragmentation that would be otherwise caused by the buddy system;
- The caching of commonly used objects so that the system does not waste time allocating, initialising and destroying objects. Benchmarks on Solaris showed excellent speed improvements for allocations with the slab allocator in use[Bon94];
- The better utilisation of hardware cache by aligning objects to the L1 or L2 caches.

Using ***vmstat****(8)* (as root) to look up slab cache information:
From man vmstat:

...

FIELD DESCRIPTION FOR SLAB MODE
    cache: Cache name
    num: Number of currently active objects
    total: Total number of available objects
    size: Size of each object
    pages: Number of pages with at least one active object
    totpages: Total number of allocated pages
    pslab: Number of pages per slab
...

<<
$ man 5 slabinfo

```
The statistics are as follows:

        active_objs
               The number of objects that are currently active (i.e., in
               use).

        num_objs
               The total number of allocated objects (i.e., objects that are
               both in use and not in use).

        objsize
               The size of objects in this slab, in bytes.

        objperslab
               The number of objects stored in each slab.

        pagesperslab
               The number of pages allocated for each slab.

 ...
```

>>

<< Below, 'Num' is the same as `active_objs` and Total the same as `num_objs >>`

```
$ sudo vmstat -m
Password: xxx
Cache                        Num  Total   Size  Pages
RAWv6                         11     11    704     11
UDPv6                         12     12    640      6
TCPv6                         12     12   1344      6
ext3_inode_cache          200585 200608    488      8
ext3_xattr                     0      0     48     85
journal_handle               340    340     24    170
revoke_record                512    512     16    256
```

```
kmalloc_dma-512              8     16    512      8
scsi_io_context              0      0    104     39

--snip--

sighand_cache              155    174   1344      6
task_struct                255    290   1440      5
...
kmalloc-2048               405    408   2048      4
kmalloc-1024               357    368   1024      4
kmalloc-512               1613   1728    512      8
kmalloc-256                115    144    256     16
kmalloc-128                777    992    128     32
kmalloc-64                5775   6080     64     64
kmalloc-32                2283   2560     32    128
kmalloc-16                3472   4096     16    256
kmalloc-8                 6021   6144      8    512
kmalloc-192               8410   9177    192     21
kmalloc-96                 761    840     96     42
$
$ sudo vmstat -m|grep task_struct
task_struct                254    290   1440      5
$ sudo vmstat -m|grep mm_struct
mm_struct                  973   1026    448      9
$ sudo vmstat -m |grep skbuff
skbuff_fclone_cache         36     36    448     18
$
```

slabtop(1) (if installed) can be used to display kernel slab cache information in real time.

Also, in recent kernels, one can view extremely detailed slab statistics exported to userspace via sysfs, particularly, */sys/kernel/slab*. For example:

```
# ls -lF /sys/kernel/slab/task_struct
lrwxrwxrwx 1 root root 0 2008-07-10 12:25 /sys/slab/task_struct -> ../slab/:0001328/
# cat /sys/kernel/slab/task_struct/order
1
#
```

Aside

Where is the socket buffer cache (it's a key networking data structure)?
See the code here: *net/core/skbuff.c:skb_init()*
Name: skbuff_head_cache

It does not show up in *vmstat -m* ? This is as 'vmstat -m' uses /proc/slabinfo to collect information on slab caches; unless CONFIG_SLUB_STATS is On, the skbuff slab cache does not show up under /proc/slabinfo.

[But it does show up under sysfs:
/sys/kernel/slab/skbuff_head_cache/
]

Memory (De)Allocation Kernel API

Low Level Page (De)Allocation

Low-Level Page Allocations Methods

API	***Description***
`struct page * alloc_page(gfp_mask)`	Allocate a single page and return a pointer to its *page* structure
`struct page * alloc_pages(gfp_mask, order)`	Allocate 2^{order} pages and return a pointer to the first page's *page* structure
`unsigned long __get_free_page(gfp_mask)`	Allocate a single page and return a pointer to its logical address
`unsigned long __get_free_pages(gfp_mask, order)`	Allocate 2^{order} pages and return a pointer to the first page's logical address
`unsigned long get_zeroed_page(gfp_mask)`	Allocate a single page, zero its contents, and return a pointer to its logical address.
`void * alloc_pages_exact(size, gfp_mask)`	Ver 2.6.27 onwards: allocates the minimum number of pages to satisfy the request

Note that both alloc_page and alloc_pages APIs return a page pointer (i.e. struct page *). To convert the same to a logical (virtual) address, use:

```
void * page_address(struct page *page);
```

Freeing pages

A family of functions allows you to free allocated pages when you no longer need them:

```
void __free_pages(struct page *page, unsigned int order);
void free_pages(unsigned long addr, unsigned int order);
void free_page(unsigned long addr);

void free_pages_exact(void *virt, size_t size);
```

You must be careful to free only pages you allocate. Passing the wrong struct page or address, or the incorrect order, can result in corruption. Remember, the kernel trusts itself. Unlike user-space, the kernel happily hangs itself if you ask it.

Let's look at an example. Here, we want to allocate eight pages:

```
unsigned long page;
page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
        /* insufficient memory: you must handle this error! */
        return -ENOMEM;
}

/* 'page' is now the address of the first of eight contiguous pages ...
*/
free_pages(page, 3);

/*
 * our pages are now freed and we should no
 * longer access the address stored in 'page'
 */
```

The `GFP_KERNEL` parameter is an example of a gfp_mask flag. It is discussed shortly.

Security: Inadvertent Information Disclosure: *__get_free_pages / free_pages* is fast; it neither initializes nor clears memory; this implies that the developer must explicitly initialize just allocated and clear just freed memory; the latter is critical from a security viewpoint as failure to clear memory risks leaving it accessible to user or kernel-space malware.

kmalloc

The kmalloc() function's operation is very similar to that of user-space's familiar malloc() routine, with the exception of the addition of a flags parameter. The kmalloc() function is a simple interface for obtaining kernel memory in byte-sized chunks. If you need whole pages, the previously discussed interfaces might be a better choice. For most kernel allocations, however, kmalloc() is the preferred interface. The function is declared in <*linux/slab.h*>:

```
void * kmalloc(size_t size, int flags);
```

<<
kzalloc() - allocate memory and set to zero:

```
void * kzalloc(size_t size, int flags)
```

```
/**
 * kzalloc - allocate memory. The memory is set to zero.
 * @size: how many bytes of memory are required.
 * @flags: the type of memory to allocate (see kmalloc).
 */
static inline void *kzalloc(size_t size, gfp_t flags)
{
    return kmalloc(size, flags | __GFP_ZERO);
}
```

>>

The function returns a pointer to a region of memory that is *at least* size bytes in length [3]. The region of memory allocated is physically contiguous. On error, it returns NULL. Kernel allocations always succeed, unless there is an insufficient amount of memory available. Thus, you must check for NULL after all calls to `kmalloc()` and handle the error appropriately.

[3] It may allocate more than you asked, although you have no way of knowing how much more! Because at its heart the kernel allocator is page-based, some allocations may be rounded up to fit within the available memory. The kernel never returns less memory than requested. If the kernel is unable to find at least the requested amount, the allocation fails and the function returns NULL.

Also, there is an upper limit (arch-specific) on the maximum number of bytes that can be allocated by kmalloc in a single call. On most architectures, the upper limit is 128 Kb*. Attempting to allocate more than this will cause kmalloc to fail.

**** Upper Limits on kmalloc() , vmalloc() API***
128 Kb is an older limit – upto and including the 2.6.21 kernel. From 2.6.22 onward, the upper limit (number of bytes that can be allocated in a single kmalloc request), is a function of the processor – well, more accurately, the page size – and the number of buddy system freelists (MAX_ORDER). On both x86 and ARM, with a standard page size of 4 Kb and MAX_ORDER of 11, the kmalloc upper limit is 4 MB! The vmalloc upper limit is, in theory, the amount of physical RAM on the system. n practice, it's usually a lot less. Typically on 32bit systems, vmalloc is severely limited by its virtual memory area. On a 32bit x86 machine, with 1GB RAM or more, vmalloc is limited to 128MB (for all allocations together, not just for one). A detailed article describing the same: kmalloc and vmalloc : Linux kernel memory allocation API Limits *The kernel module (can download the source code from the above link),also serves as a decent example of writing pure kernel code.*

<<
Also, recall that upon 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)

The kmalloc() allocations occur only *from this identity-mapped region. Thus:*
- *the allocat*ed memory *is guaranteed to be physically contiguous*
- *of small size (max is typically 4 MB)*
- *very fast- no (additional) page-table setup, etc.*

More detail (well it's really quite IA-32 specific):
"How does the linux kernel manage less than 1GB physical memory?"
>>

Let's look at an example. Assume you need to dynamically allocate enough room for a fictional dog structure:

```
struct dog *ptr;

ptr = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!ptr)
        /* handle error ... */
```

If the kmalloc() call succeeds, ptr now points to a block of memory that is at least the requested size. The GFP_KERNEL flag specifies the behavior of the memory allocator while trying to obtain the memory to return to the caller of kmalloc().

Interestingly, the kmalloc() interface is built on top of the slab layer, using a family of general purpose caches.

To summarize, advantages of kmalloc:

- allocation and de-allocation is extremely fast (slab-cached)
- fragments of RAM can be efficiently allocated
- allocated RAM is guaranteed to be physically contiguous
- kmalloc always allocates cache-line aligned memory (typically 32 bytes-aligned on x86 and ARM).

Limitations of kmalloc:

- Security: Inadvertent Information Disclosure: kmalloc / kfree is fast; it neither initializes nor clears memory; this implies that the developer must explicitly initialize just allocated and clear just freed memory; the latter is critical from a security viewpoint as failure to clear memory risks leaving it accessible to user or kernel-space malware.

  <<
  NOTE-
  - Pl try out the kernel module gfp_memclear that tests exactly this case
  - The kernel configurable CONFIG_PAGE_POISONING is meant to turn the uaf-poison feature on (uaf = use-after-free):

    Kernel Hacking / Memory Debugging / Poison pages after freeing
    CONFIG_PAGE_POISONING:

    > Fill the pages with poison patterns after free_pages() and verify
    > the patterns before alloc_pages. The filling of the memory helps
    > reduce the risk of information leaks from freed data. This does
    > have a potential performance impact.
    >
    > Note that "poison" here is not the same thing as the "HWPoison"
    > for CONFIG_MEMORY_FAILURE. This is software poisoning only.
    >
    > If unsure, say N

  ...
  \>>

- Upper limit that can be allocated in a single shot is a function of the page size and buddy system allocator configuration (MAX_ORDER); pratically speaking, on most systems – with a page size of 4 KB and a MAX_ORDER of 11, the upper limit is 4 MB.

[OPTIONAL]
<< The Instructor can skip ahead to the pertitent flags >>

gfp_mask Flags

You've seen various examples of allocator flags in both the low-level page allocation functions and kmalloc(). Now it's time to discuss these flags in depth.

The flags are broken up into three categories: action modifiers, zone modifiers, and types.

Action modifiers *specify how* the kernel is supposed to allocate the requested memory. In certain situations, only certain methods can be employed to allocate memory. For example, interrupt handlers must instruct the kernel not to sleep (because interrupt handlers cannot reschedule) in the course of allocating memory.

Zone modifiers *specify from where* to allocate memory. As you saw earlier in this chapter, the kernel divides physical memory into multiple zones, each of which serves a different purpose.

Zone modifiers specify *from which of these zones* to allocate. Type flags specify a combination of action and zone modifiers as needed by a certain type of memory allocation. Type flags simplify specifying numerous modifiers; instead, you generally specify just one type flag. The GFP_KERNEL is a type flag, which is used for code in process context inside the kernel. Let's look at the flags.

Action Modifiers

All the flags, the action modifiers included, are declared in *<linux/gfp.h>*. The file *<linux/slab.h>* includes this header, however, so you often need not include it directly. In reality, you will usually use only the type modifiers, which are discussed later. Nonetheless, it is good to have an understanding of these individual flags. Table 11.3 is a list of the action modifiers.

Action Modifiers

Flag	*Description*
__GFP_WAIT	The allocator can sleep. (GFP_KERNEL)
__GFP_HIGH	The allocator can access emergency pools. (GFP_ATOMIC)
__GFP_IO	The allocator can start disk I/O. (GFP_KERNEL)
__GFP_FS	The allocator can start filesystem I/O. (GFP_KERNEL)
__GFP_COLD	The allocator should use cache cold pages.
__GFP_NOWARN	The allocator will not print failure warnings.
__GFP_REPEAT	The allocator will repeat the allocation if it fails, but the allocation can potentially fail.

__GFP_NOFAIL	The allocator will indefinitely repeat the allocation. The allocation cannot fail.
__GFP_NORETRY	The allocator will never retry if the allocation fails.
__GFP_NO_GROW	Used internally by the slab layer.
__GFP_COMP	Add compound page metadata. Used internally by the hugetlb code.

These allocations can be specified together. For example,

```
ptr = kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);
```

instructs the page allocator (ultimately alloc_pages()) that the allocation can block, perform I/O, and perform filesystem operations, if needed. This allows the kernel great freedom in how it can find the free memory to satisfy the allocation.Most allocations specify these modifiers, but do so indirectly by way of the type flags we will discuss shortly. Don't worry - you won't have to figure out which of these weird flags to use every time you allocate memory!

Zone Modifiers

Flag	*Description*
__GFP_DMA	Allocate only from ZONE_DMA
__GFP_HIGHMEM	Allocate from ZONE_HIGHMEM or ZONE_NORMAL

--snip--
<< Details: refer the LKD3 book. >>

Type Flags

The type flags specify the required action and zone modifiers to fulfill a particular type of transaction. Therefore, kernel code tends to use the correct type flag and not specify the myriad of other flags it might need. This is both simpler and less error prone. Table 11.5 is a list of the type flags and Table 11.6 shows which modifiers are associated with each type flag.Table 11.5.

Type Flags

Flag	*Description*
GFP_ATOMIC	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spinlock, and in other situations where you cannot sleep.
GFP_NOIO	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when you cannot cause more disk I/O,

	which might lead to some unpleasant recursion.
GFP_NOFS	This allocation can block and can initiate disk I/O, if it must, but will not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.
GFP_KERNEL	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to in order to obtain the memory requested by the caller. This flag should be your first choice.
GFP_USER	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
GFP_HIGHUSER	This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.
GFP_DMA	This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the above.

Table 11.6. Listing of the Modifiers Behind Each Type Flag

Flag	*Modifier Flags*
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	(__GFP_WAIT | __GFP_IO)
GFP_KERNEL	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_USER	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_HIGHUSER	(__GFP_WAIT | __GFP_IO | __GFP_FS |__GFP_HIGHMEM)
GFP_DMA	__GFP_DMA

<<

Why all these flags?

The (partial) answer from https://lwn.net/Articles/635354/

…

The kernel's memory-management subsystem is charged with ensuring that memory is available when it is needed by either the kernel or a user-space process. That job is easy when a lot of memory is free, but it gets harder once memory fills up — as it inevitably does.

When memory gets tight and somebody is requesting more, the kernel has a couple of options:
(1) free some memory currently in use elsewhere, or
(2) deny (fail) the allocation request.

The process of freeing (or "reclaiming") memory may involve writing the current contents of that memory to persistent storage. That, in turn, involves calling into the filesystem or block I/O code. But if any of those subsystems are, in fact, the source of the allocation request, calling back into them can lead to deadlocks and other unfortunate situations.

For that reason (among others), allocation requests carry a set of flags describing the actions that can be performed in the handling of the request. The two flags of interest in this article are GFP_NOFS (calls back into filesystems are not allowed), and GFP_NOIO (no type of I/O can be started). The former inhibits attempts to write dirty pages back to files on disk; the latter can block activity like writing pages to swap.

...
>>

Let's look at the frequently used flags and when and why you might need them. The vast majority of allocations in the kernel use the GFP_KERNEL flag. The resulting allocation is a normal priority allocation that might sleep. Because the call can block, this flag can be used only from process context that can safely reschedule (that is, no locks are held and so on). Because this flag does not make any stipulations as to how the kernel may obtain the requested memory, the memory allocation has a high probability of succeeding.

On the far other end of the spectrum is the GFP_ATOMIC flag. Because this flag specifies a memory allocation that cannot sleep, the allocation is very restrictive in the memory it can obtain for the caller. If no sufficiently sized contiguous chunk of memory is available, the kernel is not very likely to free memory because it cannot put the caller to sleep. Conversely, the GFP_KERNEL allocation can put the caller to sleep to swap inactive pages to disk, flush dirty pages to disk, and so on. Because GFP_ATOMIC is unable to perform any of these actions, it has less of a chance of succeeding (at least when memory is low) compared to GFP_KERNEL allocations. Nonetheless, the GFP_ATOMIC flag is the only option when the current code is unable to sleep, such as with interrupt handlers, softirqs, and tasklets.

--*snip*--

In the vast majority of the code that you write you will use either GFP_KERNEL or GFP_ATOMIC. Table 11.7 is a list of the common situations and the flags to use. Regardless of the allocation type, you must check for and handle failures.

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
Need DMA-able memory, can sleep	Use (GFP_DMA | GFP_KERNEL)
Need DMA-able memory, cannot sleep	Use (GFP_DMA | GFP_ATOMIC), or perform your allocation at an earlier point when you can sleep

kfree

The other end of kmalloc() is kfree(), which is declared in <*linux/slab.h*>:

```
void kfree(const void *ptr);
```

The kfree() method frees a block of memory previously allocated with kmalloc(). Calling this function on memory not previously allocated with kmalloc(), or on memory which has already been freed, results in very bad things, such as freeing memory belonging to another part of the kernel. Just as in user-space, be careful to balance your allocations with your deallocations to prevent memory leaks and other bugs. Note, calling kfree(NULL) is explicitly checked for and safe.

Look at an example of allocating memory in an interrupt handler. In this example, an interrupt handler wants to allocate a buffer to hold incoming data. The preprocessor macro BUF_SIZE is the size in bytes of this desired buffer, which is presumably larger than just a couple of bytes.

```
char *buf = NULL;
buf = kmalloc(BUF_SIZE, GFP_ATOMIC);
if (!buf)
        /* error allocting memory ! */
```

Later, when you no longer need the memory, do not forget to free it:

```
kfree(buf);
```

SIDEBAR :: Small Allocations Never Fail !?

LWN : Memory management when failure is not an option , Jon Corbet, March 2015

Last December, a discussion of system stalls related to low-memory situations led to the revelation that small memory allocations never fail in the kernel. Since then, the discussion on how to best handle low-memory situations has continued, focusing in particular on situations where the kernel cannot afford to let a memory allocation fail. That discussion has exposed some significant differences of opinion on how memory allocation should work in the kernel.
…

LWN : The "too small to fail" memory-allocation rule ,
Jon Corbet, Dec 2014

Kernel developers have long been told that, with few exceptions, attempts to allocate memory can fail if the system does not have sufficient resources. As a result, in well-written code, every call to a function like kmalloc(), vmalloc(), or __get_free_pages() is accompanied by carefully thought-out error-handling code.

It turns out, though, the behavior actually implemented in the memory-management subsystem is a bit different from what is written in the brochure. That difference can lead to unfortunate run-time behavior, but the fix might just be worse.
…

When asked about this problem, XFS maintainer Dave Chinner quickly wondered why the memory-management code was resorting to the OOM killer rather than just failing the problematic memory allocation. The XFS code, he said, is nicely prepared to deal with an allocation failure; to him, using that code seems better than killing random processes and locking up the system as a whole. That is when memory management maintainer Michal Hocko dropped a bomb by saying:

Well, it has been an unwritten rule that GFP_KERNEL allocations for low-order (<=PAGE_ALLOC_COSTLY_ORDER) never fail. This is a long ago decision which would be tricky to fix now without silently breaking a lot of code. Sad...

```
<<
<linux/mmzone.h>
…
/*
 * PAGE_ALLOC_COSTLY_ORDER is the order at which allocations are deemed
 * costly to service.  That is between allocation orders which should
 * coalesce naturally under reasonable reclaim pressure and those which
 * will not.
 */
#define PAGE_ALLOC_COSTLY_ORDER 3       << 2^3 = 8 pages >>
…
>>
```

The resulting explosion could be heard in Dave's incredulous reply:

*We have *always* been told memory allocations are not guaranteed to succeed, ever, unless __GFP_NOFAIL is set, but that's deprecated and nobody is allowed to use it any more.*

*Lots of code has dependencies on memory allocation making progress or failing for the system to work in low memory situations. The page cache is one of them, which means all filesystems have that dependency. We don't explicitly ask memory allocations to fail, we *expect* the memory allocation failures will occur in low memory conditions. We've been designing and writing code with this in mind for the past 15 years.*

…

But it is worse than that: since small allocations do not fail, almost none of the thousands of error-recovery paths in the kernel now are ever exercised. They could be tested if developers were to make use of the the kernel's fault injection framework, but, in practice, it seems that few developers do so. So those error-recovery paths are not just unused and subject to bit rot; chances are that a discouragingly large portion of them have never been tested in the first place.

If the unwritten "too small to fail" rule were to be repealed, all of those error-recovery paths would become live code for the first time. In a sense, the kernel would gain thousands of lines of untested code that only run in rare circumstances where things are already going wrong. There can be no doubt that a number of obscure bugs and potential security problems would result.

That leaves memory-management developers in a bit of a bind. Causing memory allocation functions to behave as advertised seems certain to introduce difficult-to-debug problems into the kernel. But the status quo has downsides of its own, and they could get worse as kernel locking becomes more complicated. It also wastes the considerable development time that goes toward the creation of error-recovery code that will never be executed. Even so, introducing low-order memory-allocation failures at this late date may well prove too scary to be attempted, even if the long-term result would be a better kernel.

IMP :: Slab Allocation – Actual Size Limitations

The slab allocator is as fast as is possible- (de)allocations are in fact extremely fast. Also, caching "objects" - commonly or heavily used data structures - is a great idea.

Secondly, it dramatically reduces the internal fragmentation issue faced by the buddy system allocator by caching "fragments" - small sizes, much less than a page in size. This lets frequent small (de)allocations be serviced with high efficiency.

For example, on an x86 running Ubuntu 12.04 with kernel ver 3.2.0-31-generic-pae, the slab cache size is as low as 8 bytes!

This implies that doing a kmalloc for 50 bytes would get you a memory chunk 64 bytes in size! So, the wastage is almost insignificant.

However, what must be realized as well, is that when allocation requests get larger than a page in size, because the available slab caches are fixed, the wastage increases. For example, again on an x86:

```
# cat /proc/slabinfo | cut -f1 -d' '
...
kmalloc-8192
kmalloc-4096
kmalloc-2048
kmalloc-1024
kmalloc-512
kmalloc-256
kmalloc-128
kmalloc-64
kmalloc-32
kmalloc-16
kmalloc-8
kmalloc-192
kmalloc-96
...
#
```

This implies that doing a kmalloc for 2050 bytes would get you a memory chunk of not less than 4096 bytes!

Q. What about allocations greater than the maximum available slab cache size, which typically is a page frame?
A. In this case, The slab allocator uses the buddy system allocation API (in fact: *__get_free_pages()* → *alloc_pages()* → … → *__alloc_pages_nodemask()* ← (the "heart" of the zoned buddy allocator) to get the memory.

To see how much RAM is *actually allocated* by the slab layer when you do a *k[m|z]alloc*, use the kernel API *ksize()*.

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

objp must be a valid slab object- typically, the return value from *k[m|z]alloc()*.

[Source: the "ksize_test" kernel module]

Testing with a kernel module allocating different size chunks of memory (using the slab allocator of course), and calling the kernel *ksize* API to see actual object size, reveals the true situation (this test was run on a quad-core x86_64 box running the Seawolf VA kernel ver 4.2.0-42-generic):

```
# make
make -C /lib/modules/4.4.0-62-generic/build  M=/home/seawolf/0k_work/ksz_test
modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-62-generic'
Building for ARCH x86 and KERNELRELEASE 4.4.0-62-generic

[...]

make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-62-generic'
# modinfo ./ksz_test.ko
filename:       <...>/ksz_test/./ksz_test.ko
description:    A quick kmalloc/ksize test; shows how kernel memory can get
wasted by using poorly thought-out amounts
author:         Kaiwan NB, kaiwanTECH
license:        Dual MIT/GPL
srcversion:     0EA1A7179D9ECB5B48B9E8D
depends:
vermagic:       4.4.0-62-generic SMP mod_unload modversions
parm:           stepsz:Number of bytes to increment each kmalloc attempt by
(default=10000) (int)
#
# insmod ./ksz_test.ko
insmod: ERROR: could not insert module ./ksz_test.ko: Cannot allocate memory
#
```

The kernel module will probably appear to fail on insmod with a message similar to that seen above. This is expected- we stress the system asking for larger and larger chunks of kernel direct-mapped RAM via the kmalloc, we get them, don't free them and thus they run out, so ultimately it will fail.
Please LOOK UP the kernel ring buffer via 'dmesg' to see the printk's!

```
# alias dmesg
alias dmesg='/bin/dmesg --human --decode --reltime --nopager'
# dmesg
kern  :info  : [May10 12:12] [1] kmalloc(n)   :[2]actual: wastage : %age
                                bytes      :   bytes :  [2-1]  : waste
kern  :info  : [  +0.000013] kmalloc(    100) :     128 :      28 :  28%
kern  :info  : [  +0.000008] kmalloc(  10100) :   16384 :    6284 :  62%
kern  :info  : [  +0.000006] kmalloc(  20100) :   32768 :   12668 :  63%
kern  :info  : [  +0.000005] kmalloc(  30100) :   32768 :    2668 :   8%
kern  :info  : [  +0.000005] kmalloc(  40100) :   65536 :   25436 :  63%
kern  :info  : [  +0.000004] kmalloc(  50100) :   65536 :   15436 :  30%
kern  :info  : [  +0.000005] kmalloc(  60100) :   65536 :    5436 :   9%
kern  :info  : [  +0.000005] kmalloc(  70100) :  131072 :   60972 :  86%

[...]

kern  :info  : [  +0.000006] kmalloc( 490100) :  524288 :   34188 :   6%
```

```
kern  :info  : [  +0.000007] kmalloc( 500100) :  524288 :   24188 :   4%
kern  :info  : [  +0.000006] kmalloc( 510100) :  524288 :   14188 :   2%
kern  :info  : [  +0.000006] kmalloc( 520100) :  524288 :    4188 :   0%
kern  :info  : [  +0.000009] kmalloc( 530100) : 1048576 :  518476 :  97%
kern  :info  : [  +0.000009] kmalloc( 540100) : 1048576 :  508476 :  94%
kern  :info  : [  +0.000009] kmalloc( 550100) : 1048576 :  498476 :  90%
kern  :info  : [  +0.000008] kmalloc( 560100) : 1048576 :  488476 :  87%

[...]

kern  :info  : [  +0.000009] kmalloc(1010100) : 1048576 :   38476 :   3%
kern  :info  : [  +0.000009] kmalloc(1020100) : 1048576 :   28476 :   2%
kern  :info  : [  +0.000009] kmalloc(1030100) : 1048576 :   18476 :   1%
kern  :info  : [  +0.000008] kmalloc(1040100) : 1048576 :    8476 :   0%
kern  :info  : [  +0.000014] kmalloc(1050100) : 2097152 : 1047052 :  99%
kern  :info  : [  +0.000015] kmalloc(1060100) : 2097152 : 1037052 :  97%
kern  :info  : [  +0.000015] kmalloc(1070100) : 2097152 : 1027052 :  95%

[...]


kern  :info  : [  +0.000014] kmalloc(2040100) : 2097152 :   57052 :   2%
kern  :info  : [  +0.000015] kmalloc(2050100) : 2097152 :   47052 :   2%
kern  :info  : [  +0.000014] kmalloc(2060100) : 2097152 :   37052 :   1%
kern  :info  : [  +0.000015] kmalloc(2070100) : 2097152 :   27052 :   1%
kern  :info  : [  +0.000014] kmalloc(2080100) : 2097152 :   17052 :   0%
kern  :info  : [  +0.000014] kmalloc(2090100) : 2097152 :    7052 :   0%
kern  :info  : [  +0.000026] kmalloc(2100100) : 4194304 : 2094204 :  99%
kern  :info  : [  +0.000027] kmalloc(2110100) : 4194304 : 2084204 :  98%
kern  :info  : [  +0.000028] kmalloc(2120100) : 4194304 : 2074204 :  97%
kern  :info  : [  +0.000028] kmalloc(2130100) : 4194304 : 2064204 :  96%
kern  :info  : [  +0.000028] kmalloc(2140100) : 4194304 : 2054204 :  95%

[...]

kern  :info  : [  +0.000027] kmalloc(4140100) : 4194304 :   54204 :   1%
kern  :info  : [  +0.000027] kmalloc(4150100) : 4194304 :   44204 :   1%
kern  :info  : [  +0.000027] kmalloc(4160100) : 4194304 :   34204 :   0%
kern  :info  : [  +0.000027] kmalloc(4170100) : 4194304 :   24204 :   0%
kern  :info  : [  +0.000027] kmalloc(4180100) : 4194304 :   14204 :   0%
kern  :info  : [  +0.000028] kmalloc(4190100) : 4194304 :    4204 :   0%
kern  :alert : [  +0.000014] kmalloc fail, num=4200100
#
```

So, we conclude that for small requests the slab allocator is very efficient, wastage is very low. But for large allocations (large meaning greater than the maximum available slab cache size, which typically is just 1 or 2 pages) it's just as bad as the buddy system? What one must realize is: for such allocations, it *is* in reality the buddy system allocator making the allocation!

Moral: allocate just how much you need, no more, and as far as possible using rounded page-sizes. If you have a real use-case of an inconveniently-sized but frequently allocated and de-allocated data structure, create a custom slab cache for it!

A Solution! "Exact" Pages Allocations

From kernel ver 2.6.27.
Github tree commit.

From the patch:
" [PATCH] Add alloc_pages_exact() and free_pages_exact()
alloc_pages_exact() is similar to alloc_pages(), except that it allocates
the minimum number of pages to fulfill the request. This is useful if you
want to allocate a very large buffer that is slightly larger than an
even power-of-two number of pages. In that case, alloc_pages() will waste
a lot of memory.

Signed-off-by: Timur Tabi <timur [at] freescale>

I have a video driver that wants to allocate a 5MB buffer. alloc_pages()
will waste 3MB of physically-contiguous memory. Therefore, I would
like to see alloc_pages_exact() added to 2.6.27.
..."

mm/page_alloc.c

```
/**
 * alloc_pages_exact - allocate an exact number physically-contiguous pages.
 * @size: the number of bytes to allocate
 * @gfp_mask: GFP flags for the allocation
 *
 * This function is similar to alloc_pages(), except that it allocates the
 * minimum number of pages to satisfy the request.  alloc_pages() can only
 * allocate memory in power-of-two pages.
 *
 * This function is also limited by MAX_ORDER.
 *
 * Memory allocated by this function must be released by free_pages_exact().
 */
void *alloc_pages_exact(size_t size, gfp_t gfp_mask)
{
...
```

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

Also, for NUMA:

```
/**
 * alloc_pages_exact_nid - allocate an exact number of physically-contiguous
 *                                pages on a node.
 * @nid: the preferred node ID where memory should be allocated
```

```
 * @size: the number of bytes to allocate
 * @gfp_mask: GFP flags for the allocation
 *
 * Like alloc_pages_exact(), but try to allocate on node nid first before falling
 * back.
 */
void * __meminit alloc_pages_exact_nid(int nid, size_t size, gfp_t gfp_mask)
...
```

[OPTIONAL / FYI]

Also, one should note that there are several “flavours” of “slab” allocator algorithms.
From the kernel configuration help:
…
SLAB

CONFIG_SLAB:
The regular slab allocator that is established and known to work well in all environments. It organizes cache hot objects in per cpu and per node queues.

…

SLUB (Unqueued Allocator)

CONFIG_SLUB:
SLUB is a slab allocator that minimizes cache line usage instead of managing queues of cached objects (SLAB approach). Per cpu caching is realized using slabs of objects instead of queues of objects. SLUB can use memory efficiently and has enhanced diagnostics. SLUB is the default choice for a slab allocator.

…

SLOB (Simple Allocator)

CONFIG_SLOB:
SLOB replaces the stock allocator with a drastically simpler allocator. SLOB is generally more space efficient but does not perform as well on large systems.

Additional Resources

SLUB on Wikipedia

The SLUB Allocator, LWN, April 2007

Toward a more efficient slab allocator, Jon Corbet, LWN, Jan 2015

"Following up on Jesper Brouer's session on networking performance, Christoph Lameter's LCA kernel miniconf session covered ways in which the performance of the kernel's low-level object allocators (the "slab" allocators) could be improved to meet current and future demands. Some of the work he covered is new, but some of it has been around, in concept at least, for some time. …"

---

vmalloc

The vmalloc() function works in a similar fashion to kmalloc(), except it allocates memory that is only virtually contiguous and not necessarily physically contiguous. This is how a user-space allocation function works: The pages returned by malloc() are contiguous within the virtual address space of the processor, but there is no guarantee that they are actually contiguous in physical RAM. The kmalloc() function guarantees that the pages are physically contiguous (and virtually contiguous).

The vmalloc() function only ensures that the pages are contiguous within the virtual address space. It does this by allocating potentially noncontiguous chunks of physical memory and "fixing up" the page tables to map the memory into a contiguous chunk of the logical address space.

<<

The (virtual) address range used by *kmalloc* and *_ _get_free_pages* features a one-to-one mapping to physical memory, possibly shifted by a constant PAGE_OFFSET value; the functions don't need to modify the page tables for that address range. The address range used by *vmalloc* and *ioremap*, on the other hand, is completely synthetic, and each allocation builds the (virtual) memory area by suitably setting up the page tables.

This difference can be perceived by comparing the pointers returned by the allocation functions. ... Addresses available for *vmalloc* are in the range from VMALLOC_START to VMALLOC_END.

On a 32-bit x86 running Ubuntu 12.10 kernel ver 3.5.0-23-generic :

```
PAGE_OFFSET   = 0xc0000000 TASK_SIZE=0xc0000000
VMALLOC_START = 0xf83fe000 VMALLOC_END=0xffbfe000    << 120 MB >>
MODULES_VADDR = 0xf83fe000 MODULES_END=0xffbfe000
```

Recommendation (and Assignment)
Write a simple kernel module to print out the above values, and the page size, on your target board or system.
>>

For the most part, only hardware devices require physically contiguous memory allocations. On many architectures, hardware devices live on the other side of the memory management unit and, thus, do not understand virtual addresses. Consequently, any regions of memory that hardware devices work with must exist as a physically contiguous block and not merely a virtually contiguous one. Blocks of memory used only by software - for example, process-related buffers - are fine using memory that is only virtually contiguous. In your programming, you will never know the difference. All memory appears to the kernel as logically contiguous.

<<
…

(vmalloc) memory is not allocated from the buddy system in a single chunk but page-by-page. This is a key aspect of vmalloc . If it were certain that a contiguous allocation could be made, there would be no need to use vmalloc .

After all, the whole purpose of the function is to reserve large memory chunks even though they may not be contiguous owing to fragmentation of the available memory. Splitting the allocation into the smallest possible units — in other words, individual pages — ensures that vmalloc will still work even when physical memory is fragmented.

...

Three physical pages whose (fictitious) positions in RAM are 1,023, 725 and 7,311 are mapped one after the other. In the virtual vmalloc area, the kernel sees them as a contiguous memory area starting at the VMALLOC_START + 100 .

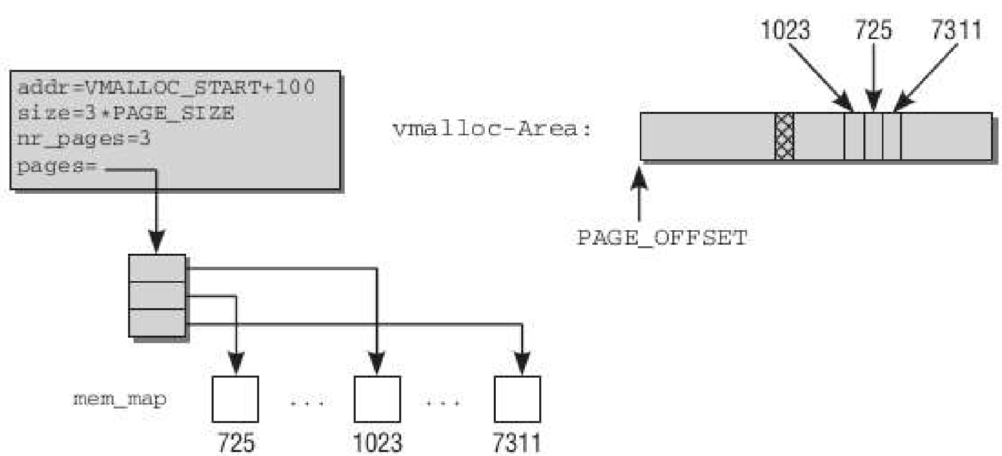


Figure 3-38: Mapping physical pages into the `vmalloc` area.

Source: PLKA
>>

Despite the fact that physically contiguous memory is required in only certain cases, most kernel code uses kmalloc() and not vmalloc() to obtain memory. Primarily, this is for performance.

The vmalloc() function, to make nonphysically contiguous pages contiguous in the virtual address space, must specifically set up the page table entries. Worse, pages obtained via vmalloc() must be mapped by their individual pages (because they are not physically contiguous), which results in much greater TLB[4] thrashing than you see when directly mapped memory is used. Because of these concerns, vmalloc() is used only when absolutely necessary - typically, to obtain very large regions of memory. For example, when modules are dynamically inserted into the kernel, they are loaded into memory created via vmalloc().

[4] The TLB (translation lookaside buffer) is a hardware cache used by most architectures to cache the mapping of virtual addresses to physical addresses. This greatly improves the performance of the system, because most memory access is done via virtual addressing.

The vmalloc() function is declared in <linux/vmalloc.h> and defined in mm/vmalloc.c. Usage is identical to user-space's malloc():

```
void * vmalloc(unsigned long size);
```

The function returns a pointer to at least size bytes of virtually contiguous memory. On error, the function returns NULL. The function might sleep, and thus cannot be called from interrupt context or other situations where blocking is not permissible.

To free an allocation obtained via vmalloc(), use

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

This function frees the block of memory beginning at addr that was previously allocated via vmalloc(). The function can also sleep and, thus, cannot be called from interrupt context. It has no return value.

Usage of these functions is simple:

```
char *buf=NULL;
buf = vmalloc(16 * PAGE_SIZE); /* get 16 pages */
if (!buf)
        /* error! failed to allocate memory */

/*
 * buf now points to at least a 16*PAGE_SIZE bytes
 * of virtually contiguous block of memory
 */
```

After you are finished with the memory, make sure to free it by using

```
vfree(buf);
```

<<
Source
…
Large contiguous memory allocations can be unreliable in the Linux kernel.

Kernel programmers will sometimes respond to this problem by allocating pages with vmalloc(). This solution (is) not ideal, though. On 32-bit systems, memory from vmalloc() must be mapped into a relatively small address space; it's easy to run out. On SMP systems, the page table changes required by vmalloc() allocations can require expensive cross-processor interrupts on all CPUs. And, on all systems, use of space in the vmalloc() range increases pressure on the translation lookaside buffer (TLB), reducing the performance of the system.

In many cases, the need for memory from vmalloc() can be eliminated by piecing together an array from smaller parts; the flexible array library exists to make this task easier.
…
>>

vmalloc usage information can be viewed via the procfs interface file 'vmallocinfo':

Eg.

```
ARM / > cat /proc/vmallocinfo
0xc8800000-0xc8802000    8192 smc_drv_probe+0x198/0x7c4 ioremap
0xc8802000-0xc8804000    8192 amba_kmi_probe+0xc4/0x15c ioremap
0xc8804000-0xc8806000    8192 amba_kmi_probe+0xc4/0x15c ioremap
0xc8832000-0xc8834000    8192 pl011_probe+0x88/0x198 ioremap
0xc8834000-0xc8836000    8192 pl011_probe+0x88/0x198 ioremap
0xc8836000-0xc8838000    8192 pl011_probe+0x88/0x198 ioremap
0xc8838000-0xc883a000    8192 pl011_probe+0x88/0x198 ioremap
0xc883a000-0xc8846000   49152 cramfs_uncompress_init+0x2c/0x60 pages=11
vmalloc
0xc8846000-0xc8889000  274432 jffs2_zlib_init+0x18/0xb4 pages=66 vmalloc
0xc8889000-0xc8895000   49152 jffs2_zlib_init+0x50/0xb4 pages=11 vmalloc
0xc8896000-0xc8898000    8192 clcdfb_probe+0x108/0x360 ioremap
ARM / >
```

As an aside, apparently the *ioremap()*, (seen later in 'Drivers'), internally uses the vmalloc().

FAQ: How can one allocate arbitrary sized kernel memory chunks with particular protections?

A. The comment over the kernel vmalloc() says:

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)
{
    return __vmalloc_node(size, 1, gfp_mask, prot, NUMA_NO_NODE,
                __builtin_return_address(0));
}
```

```
EXPORT_SYMBOL(__vmalloc);
```

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

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

Custom Slab Cache Interface

<< Source: "[Anatomy of the Linux Slab Allocator]" by M. Tim Jones >>
--snip--

Now on to the application program interface (API) for creating new slab caches, adding memory to the caches, destroying the caches, as well as the functions to allocate and deallocate objects from them.

The first step is to create your slab cache structure, which you can create statically as:

```
static struct kmem_cache *my_cachep;
```

This reference is then used by the other slab cache functions for creation, deletion, allocation, and so on. The kmem_cache structure contains the per-central processing unit (CPU) data, a set of tunables (accessible through the proc file system), statistics, and necessary elements to manage the slab cache.

kmem_cache_create

Kernel function kmem_cache_create is used to create a new cache. This is commonly performed at kernel init time or when a kernel module is first loaded. Its prototype is defined as:

```
struct kmem_cache *
kmem_cache_create(const char *name, unsigned int size,
                  unsigned int align,
                  slab_flags_t flags, void (*ctor)(void *));
```

The name argument defines the name of the cache, which is used by the proc file system (in /proc/slabinfo) to identify this cache. The size argument specifies the size of the objects that should be created for this cache, with the align argument defining the required alignment for each object. The flags argument specifies options to enable for the cache. These flags are shown in Table 1.

Table 1. Partial list of options for kmem_cache_create (as specified in flags)

Option	***Description***
`SLAB_RED_ZONE`	Insert markers at header and trailer of the object to support checking of buffer overruns.
`SLAB_POISON`	Fill a slab with a known pattern (0xa5a5a5a5) to allow monitoring of objects in the cache (objects owned by the cache, but modified externally). *<< Could be useful when debugging memory initialization problems. Ref:* [Hexspeak]() >>
`SLAB_HWCACHE_ALIGN`	Specify that the objects in this cache must be aligned to the hardware cachline.

<<

An FAQ: is there a way to guarantee "aligned" kernel memory on allocation?

Page-aligned memory cannot be guaranteed on allocation. (If *really* required, the technique is to allocate twice the amount you require, save the original pointer, then increment it to the first page boundary and use that, freeing the original pointer when done. Wasteful, though.)

Cache-line -aligned memory upon allocation is possible. To guarantee this (in an architecture-independent fashion), leverage the slab allocator API.
Create a custom slab cache using *kmem_cache_create* and specify SLAB_HWCACHE_ALIGN as one of the flags.

In fact, kmalloc always allocates cache-line aligned memory (typically 64 or 32 bytes-aligned on x86 and ARM).

From the [LinkedIn Linux Internals and Device Drivers Group]():
"kmalloc always allocates memory which is cache-line aligned (which is 32-bytes on x86 and ARM). For allocating memory which you need to DMA from, you're better off to use dma_alloc_coherent (for ARM). For x86 I think you may want to use a pci_xxx function instead, but I've never coded DMA for x86.

While it is possible to DMA from kmalloc'd memory, you need to more work to ensure memory coherency.
- Dave Hylands."
>>

The ctor argument define an optional object constructor. The constructor is a callback function provided by the user. When a new object is allocated from the cache, it can be initialized through the constructor. << The Linux kernel does not often make use of this; you can initialize the constructor to NULL if you don't wish to use it).>>

After the cache is created, its reference is returned by the kmem_cache_create function. Note that this function allocates no memory to the cache. Instead, when attempts are made to allocate objects from the cache (when it's initially empty), memory is allocated to it through a refill operation. This same operation is used to add memory to the cache when all its objects are consumed.

kmem_cache_destroy

Kernel function kmem_cache_destroy is used to destroy a cache. This call is performed by kernel modules when they are unloaded. The cache must be empty before this function is called.

```
void kmem_cache_destroy( struct kmem_cache *cachep );
```

kmem_cache_alloc

To allocate an object from a named cache, the kmem_cache_alloc function is used. The caller provides the cache from which to allocate an object and a set of flags:

```
void * kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

This function returns an object from the cache. Note that if the cache is currently empty, this function may invoke cache_alloc_refill to add memory to the cache. The flag options for kmem_cache_alloc are the same as those for kmalloc. Table 2 provides a partial list of the flag options.

…

kmem_cache_zalloc

The kernel function kmem_cache_zalloc is similar to kmem_cache_alloc, except that it performs a memset of the object to clear it out prior to returning the object to the caller.

kmem_cache_free

To free an object back to the slab, the kmem_cache_free is used. The caller provides the cache reference and the object to be freed.

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

Other miscellaneous functions

The slab cache API provides a number of other useful functions. The kmem_cache_size function returns the size of the objects that are managed by this cache. You can also retrieve the name of a given cache (as defined at cache creation time) through a call to kmem_cache_name. A cache can be shrunk by releasing free slabs in the cache. This can be accomplished with a call to kmem_cache_shrink. Note that this action (called reaping) is performed automatically on a periodic basis by the kernel (through kswapd).

```
unsigned int kmem_cache_size( struct kmem_cache *cachep );
const char *kmem_cache_name( struct kmem_cache *cachep );
int kmem_cache_shrink( struct kmem_cache *cachep );
```

--snip--

Example use of the slab cache

The following code snippets demonstrate the creation of a new slab cache, allocating and deallocating objects from the cache, and then destroying the cache. To begin, a kmem_cache object must be defined and then initialized (see Listing 1). This particular cache contains 32-byte objects and is hardware-cache aligned (as defined by the flags argument SLAB_HWCACHE_ALIGN).

Listing 1. Creating a new slab cache

```
#define MYBUFSZ 350         /* using size '350' seems to give
                                us a new private slab cache */
typedef struct {
      u32 a, b, c;
      s8 *cptr;
      u16 valid;
      u8 intbuf[MYBUFSZ];
} MyStruct;

static struct kmem_cache *my_cachep;

static void init_my_cache( void )
{
/*
struct kmem_cache *
kmem_cache_create( const char *name, size_t size, size_t align,
       unsigned long flags,
       void (*ctor)(void*));
*/
   my_cachep = kmem_cache_create(
                 "my_cache",            /* Name */
                 sizeof(MyStruct),      /* Object Size */
                 0,                     /* Alignment */
                 SLAB_HWCACHE_ALIGN,    /* Flags */
                 NUL);                  /* Constructor */

   return;
}
```

With your slab cache allocated, you can now allocate an object from it. Listing 2 provides an example of allocating and deallocating an object from the cache. It also demonstrates two of the miscellaneous functions.

Listing 2. Allocating and deallocating objects

```
int slab_test( void )
{
  void *object;

#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,39) // add a portability
check
      printk("Cache name is %s\n",
            kmem_cache_name(my_cachep));
#else
      printk("[ker ver > 2.6.38 cache name deprecated...]\n");
#endif
  printk("Cache object size is %d\n",
         kmem_cache_size(my_cachep));

  object = kmem_cache_alloc (my_cachep, GFP_KERNEL);

  if (object) {
    kmem_cache_free (my_cachep, object);
  }
  return 0;
}
```

Finally, Listing 3 is an example of destroying a slab cache. The caller must ensure that no attempts to allocate objects from the cache are performed during the destroy operation.

Listing 3. Destroying a slab cache

```
static void remove_my_cache( void )
{
  if (my_cachep)
    kmem_cache_destroy( my_cachep );
  return;
}
```

Quick Lookup

We use cscope (on the 4.4.0-rc6 kernel sources) to see which functions call *kmem_cache_create():*

```
Functions calling this function: kmem_cache_create

  File                  Function                        Line
0 memory.c              dlm_memory_init                   24 lkb_cache =
kmem_cache_create("dlm_lkb", sizeof(struct dlm_lkb),
```

```
1 memory.c                dlm_memory_init                 29 rsb_cache =
kmem_cache_create("dlm_rsb", sizeof(struct dlm_rsb),
2 main.c                  ecryptfs_init_kmem_caches      758 *(info->cache) =
kmem_cache_create(info->name, info->size,
3 super.c                 init_inodecache                 96 efs_inode_cachep =
kmem_cache_create("efs_inode_cache",
4 eventpoll.c             eventpoll_init                2124 epi_cache =
kmem_cache_create("eventpoll_epi", sizeof(struct epitem),
5 eventpoll.c             eventpoll_init                2128 pwq_cache =
kmem_cache_create("eventpoll_pwq",
6 super.c                 init_inodecache                195 exofs_inode_cachep =
kmem_cache_create("exofs_inode_cache",
7 super.c                 init_inodecache                203 ext2_inode_cachep =
kmem_cache_create("ext2_inode_cache",
8 extents_status.c        ext4_init_es                   154 ext4_es_cachep =
kmem_cache_create("ext4_extent_status",
9 mballoc.c               ext4_groupinfo_create_slab    2559 cachep =
kmem_cache_create(ext4_groupinfo_slab_names[cache_index],
a super.c                 init_inodecache                966 ext4_inode_cachep =
kmem_cache_create("ext4_inode_cache",
b f2fs.h                  f2fs_kmem_cache_create        1280 return kmem_cache_create(name, size,
0, SLAB_RECLAIM_ACCOUNT, NULL);
c cache.c                 fat_cache_init                  47 fat_cache_cachep =
kmem_cache_create("fat_cache",
d inode.c                 fat_init_inodecache            677 fat_inode_cachep =
kmem_cache_create("fat_inode_cache",
e fcntl.c                 fcntl_init                     754 fasync_cache =
kmem_cache_create("fasync_cache",
f file_table.c            files_init                     314 filp_cachep =
kmem_cache_create("filp", sizeof(struct file), 0,
g vxfs_super.c            vxfs_init                      269 vxfs_inode_cachep =
kmem_cache_create("vxfs_inode",
h main.c                  fscache_init                   143 fscache_cookie_jar =
kmem_cache_create("fscache_cookie_jar",
i dev.c                   fuse_dev_init                 2296 fuse_req_cachep =
kmem_cache_create("fuse_request",
j inode.c                 fuse_fs_init                  1257 fuse_inode_cachep =
kmem_cache_create("fuse_inode",
k main.c                  init_gfs2_fs                    97 gfs2_glock_cachep =
kmem_cache_create("gfs2_glock",
l main.c                  init_gfs2_fs                   104 gfs2_glock_aspace_cachep =
kmem_cache_create("gfs2_glock(aspace)",
m main.c                  init_gfs2_fs                   112 gfs2_inode_cachep =
kmem_cache_create("gfs2_inode",

* Lines 174-197 of 377, 181 more - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
…
```

Kernel Freeing of Caches under Memory Pressure

Bear in mind that caches are a performance optimization and not a necessity. Thus, the kernel can and does “request” caches to free some amount of their memory when a low memory situation arises.

The developer must therefore hook into this system via :

```
include/linux/shrinker.h
extern int register_shrinker(struct shrinker *);
extern void unregister_shrinker(struct shrinker *);
```

See this LWN Article: Smarter Shrinkers, Jon Corbet, May 2013.

In summary:

Which Allocation Method Should I Use?

Required Memory	***Method***	***API(s)***
Physically contiguous perfectly rounded power-of-2 pages	BSA / page allocator	`alloc_page[s]`, `__get_free_page[s]`, `get_zeroed_page`, `alloc_pages_exact`
Physically contiguous memory of size < 1 page; upto 4 MB max typically	Slab Allocator (cache)	`kmalloc`, `kzalloc`
Virtually contiguous memory of large size	Indirect via BSA	`vmalloc`
Custom data structures	Custom slab caches	`kmem_cache_*` APIs

- ✔ If you need contiguous physical pages, use one of the low-level page allocators or `kmalloc()`. This is the standard manner of allocating memory from within the kernel, and most likely, how you will allocate most of your memory. Recall that the two most common flags given to these functions are GFP_ATOMIC and GFP_KERNEL. Specify the GFP_ATOMIC flag to perform a high priority allocation that will not sleep. This is a requirement of interrupt handlers and other pieces of code that cannot sleep. Code that can sleep, such as process context code that does not hold a spin lock, should use GFP_KERNEL. This flag specifies an allocation that can sleep, if needed, to obtain the requested memory.

- ✔ If you want to allocate from high memory, use `alloc_pages()`. The `alloc_pages()` function returns a `struct page`, and not a pointer to a logical address. Because high memory might not be mapped, the only way to access it might be via the corresponding `struct page` structure. To obtain an actual pointer, use `kmap()` to map the high memory into the kernel's logical address space.

- ✔ >= 2.6.27 : `alloc_pages_exact()` allocates the minimum number of pages to satisfy the request

- ✔ If you do not need physically contiguous pages - only virtually contiguous - use `vmalloc()`, although bear in mind the slight performance hit taken with `vmalloc()` over `kmalloc()`. The `vmalloc()` function allocates kernel memory that is virtually contiguous but not, per se, physically contiguous. It performs this feat much as user-space allocations do, by mapping chunks of physical memory into a contiguous logical address space.

- ✔ If you are creating and destroying many large data structures, consider setting up a slab cache. The slab layer maintains a per-processor object cache (a free list), which might greatly enhance object allocation and deallocation performance. Rather than frequently allocate and free memory, the slab layer stores a cache of already allocated objects for you. When you need a new chunk of memory to hold your data structure, the slab layer often does not need to allocate more memory and instead simply can return an object from the cache.

Kernel Segment: Kernel Memory Map on the Linux OS

We use the popular **ARM microprocessor** as a reference example.
Text below reproduced from http://www.arm.linux.org.uk/developer/memory.txt :

Kernel Memory Layout on ARM Linux

Russell King <rmk@arm.linux.org.uk>
November 17, 2005 (2.6.15)

This document describes the virtual memory layout which the Linux kernel uses for ARM processors. It indicates which regions are free for platforms to use, and which are used by generic code.

The ARM CPU is capable of addressing a maximum of 4GB virtual memory space, and this must be shared between user space processes, the kernel, and hardware devices.

As the ARM architecture matures, it becomes necessary to reserve certain regions of VM space for use for new facilities; therefore this document may reserve more VM space over time.

```
Start       End         Use
----------------------------------------------------------------------
ffff8000    ffffffff    copy_user_page / clear_user_page use.
                        For SA11xx and Xscale, this is used to
                        setup a minicache mapping.

ffff1000    ffff7fff    Reserved.
                        Platforms must not use this address range.

ffff0000    ffff0fff    CPU vector page.
                        The CPU vectors are mapped here if the
                        CPU supports vector relocation (control
                        register V bit.)

ffc00000    fffeffff    DMA memory mapping region.  Memory returned
                        by the dma_alloc_xxx functions will be
                        dynamically mapped here.

ff000000    ffbfffff    Reserved for future expansion of DMA
                        mapping region.

VMALLOC_END   feffffff  Free for platform use, recommended.
                        VMALLOC_END must be aligned to a 2MB
                        boundary.
```

```
VMALLOC_START  VMALLOC_END-1  vmalloc() / ioremap() space.
                        Memory returned by vmalloc/ioremap will
                        be dynamically placed in this region.
                        VMALLOC_START may be based upon the value
                        of the high_memory variable.

PAGE_OFFSET  high_memory-1    Kernel direct-mapped RAM region.
                        This maps the platforms RAM, and typically
                        maps all platform RAM in a 1:1 relationship.

TASK_SIZE   PAGE_OFFSET-1     Kernel module space
                        Kernel modules inserted via insmod are
                        placed here using dynamic mappings.

00001000    TASK_SIZE-1 User space mappings
                        Per-thread mappings are placed here via
                        the mmap() system call.

00000000    00000fff    CPU vector page / null pointer trap
                        CPUs which do not support vector remapping
                        place their vector page here.  NULL pointer
                        dereferences by both the kernel and user
                        space are also caught via this mapping.
```

Please note that mappings which collide with the above areas may result in a non-bootable kernel, or may cause the kernel to (eventually) panic at run time.

Since future CPUs may impact the kernel mapping layout, user programs must not access any memory which is not mapped inside their 0x0001000 to TASK_SIZE address range. If they wish to access these areas, they must set up their own mappings using open() and mmap().

Article:
Understanding the VMALLOC region Overlap

SIDEBAR :: The null trap page(s) and Security !

We are given to understand that the NULL pointer – literally the 0 value – is always illegal and impossible to dereference. This is usually true, but not always!!

See this very interesting security-hacking related article on an actual proof-of-concept security breakage on the 2.6.30 Linux kernel (with SELinux installed):

“Fun with NULL pointers, part 1”, Jonanthan Corbet, LWN, July 2009.
and https://psomas.wordpress.com/2011/02/17/gimme-root/

Describes in some detail a local kernel exploit by Brad Spengler – viz. “CheddarBay” [CVE-2009-1897].

“... In well-designed systems, catastrophic failures are rarely the result of a single failure. That is certainly the case here. Several things went wrong to make this exploit possible: security modules were able to grant access to low memory mappings contrary to system policy, the SELinux policy allowed those mappings, pulseaudio can be exploited to make a specific privileged operation available to exploit code, a NULL pointer was dereferenced before being checked, the check was optimized out by the compiler, and the code used the NULL pointer in a way which allowed the attacker to take over the system.

It is a long chain of failures, each of which was necessary to make this exploit possible.
This particular vulnerability has been closed, but there will almost certainly be others like it. See the second article in this series for a look at how the kernel developers are responding to this exploit.”

<<
An Aside:
The worst mistake of computer science (the NULL, Tony Hoare)
>>

<<
Linux OS Kernel Segment ::

Eg. on a QEMU-emulated ARM (Versatile Express CA-9) Linux with 256 MB RAM, the kernel printk's on boot show:

```
...
[    0.000000] Memory: 246640k/246640k available, 15504k reserved, 0K highmem
[    0.000000] Virtual kernel memory layout:
[    0.000000]     vector  : 0xffff0000 - 0xffff1000   (   4 kB)
[    0.000000]     fixmap  : 0xfff00000 - 0xfffe0000   ( 896 kB)
[    0.000000]     vmalloc : 0xd0800000 - 0xff000000   ( 744 MB)
[    0.000000]     lowmem  : 0xc0000000 - 0xd0000000   ( 256 MB)
[    0.000000]     modules : 0xbf000000 - 0xc0000000   (  16 MB)
[    0.000000]       .text : 0xc0008000 - 0xc0631c90   (6312 kB)
[    0.000000]       .init : 0xc0632000 - 0xc0684240   ( 329 kB)
[    0.000000]       .data : 0xc0686000 - 0xc06d5e20   ( 320 kB)
[    0.000000]        .bss : 0xc06d5e20 - 0xc0c607c4   (5675 kB)
…
```

"lowmem" is the identity-mapped (directly 1:1 mapped) system RAM into kernel segment.
>>

Examining the Kernel Segment – kernel Virtual Address Space

<< Updated ver 06 July 2016 >>

```
$ cat vm_show_kernel_seg.c
/*
 * vm_show_kernel_seg.c
 *
 * Simple kernel module to show various kernel virtual addresses, including
 * some user process context virtual addresses.
 * Useful for learning, testing, etc.
 * For both 32 and 64-bit systems.
 *
 * (c) 2011-2018 Kaiwan NB, kaiwanTECH.
 * License: MIT
 */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/highmem.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include <asm/pgtable.h>
```

```
#include "convenient.h"  // adjust path as required

#if(BITS_PER_LONG == 32)
	#define FMTSPC		"%08x"
	#define FMTSPC_DEC "%08d"
	#define TYPECST		unsigned int
	#define MY_PATTERN1	0xdeadface
	#define MY_PATTERN2	0xffeeddcc
#elif(BITS_PER_LONG == 64)
	#define FMTSPC		"%016lx"
	#define FMTSPC_DEC "%ld"
	#define TYPECST		 unsigned long
	#define MY_PATTERN1	0xdeadfacedeadface
	#define MY_PATTERN2	0xffeeddccbbaa9988
#endif

static unsigned int statgul;

void bar(void)
{
	pr_info("--------------------Stack Dump:-------------------------------\
n");
	dump_stack();
	pr_info("--------------------------------------------------------------\
n");
}
void foo(void)
{
	char c='x';
	pr_info("&c = 0x" FMTSPC "\n", (TYPECST)&c);
	bar();
}

// Init mem to a pattern (like 0xdead 0xface .... ....)
static inline void mempattern(
 void *pdest,
#if(BITS_PER_LONG == 32)
 u32 pattern,
#elif(BITS_PER_LONG == 64)
 u64 pattern,
#endif
 int len)
{
	int i;
#if(BITS_PER_LONG == 32)
	u32 *pattptr = (u32 *)pdest;
	for (i=0; i < len/sizeof(u32 *); i++)
#elif(BITS_PER_LONG == 64)
	u64 *pattptr = (u64 *)pdest;
	for (i=0; i < len/sizeof(u64 *); i++)
#endif
					{
		//pr_info("pattptr=%pK\n", pattptr);
		*pattptr = pattern;
		pattptr ++;
```

```
      }
}

static void *kptr=NULL, *vptr=NULL;
volatile static    u32 vg=0x1234abcd;

static int __init vm_img_init(void)
{
   int knum=512,disp=32;

   pr_info("Platform:\n");
#ifdef CONFIG_X86
 #if(BITS_PER_LONG == 32)
      pr_info(" x86-32 : ");
 #else
      pr_info(" x86_64 : ");
 #endif
#endif
#ifdef CONFIG_ARM
      pr_info(" ARM-32 : ");
#endif
#ifdef CONFIG_ARM64
      pr_info(" ARM64 : ");
#endif
#ifdef CONFIG_MIPS
      pr_info(" MIPS : ");
#endif
#ifdef CONFIG_PPC
      pr_info(" PPC : ");
#endif

#if(BITS_PER_LONG == 32)
      pr_info (" 32-bit OS ");
#elif(BITS_PER_LONG == 64)
      pr_info (" 64-bit OS ");
#endif
#ifdef __BIG_ENDIAN  // just for the heck of it..
      pr_info(" Big-endian.\n");
#else
      pr_info(" Little-endian.\n");
#endif

      /* Ha! When using "%d" etc for sizeof(), the compiler would complain:
       * ... warning: format ‘%d’ expects argument of type ‘int’, but
       * argument 5 has type ‘long unsigned int’ [-Wformat=] ...
       * Turns out we shoud use "%zu" to correctly represent size_t (which sizeof
operator returns)!
       */
      pr_info (" sizeof(int)   =%zu, sizeof(long) =%zu\n"
             " sizeof(void *)=%zu, sizeof(u64 *)=%zu\n",
            sizeof(int), sizeof(long), sizeof(void *), sizeof(u64 *));

      kptr = kmalloc(knum, GFP_KERNEL);
      if (!kptr) {
            pr_alert("kmalloc failed!\n");
```

```
            return -ENOMEM;
      }
      /*
       * !IMP! NOTE reg the 'new' security-conscious printk formats!
       * SHORT story:
       * We need to avoid 'leaking' kernel addr to userspace (hackers
       * have a merry time!).
       * Now %p will _not_ show the actual kernel addr, but rather a
       * hashed value.
       *
       * To see the actual addr use %px : dangerous!
       * To see the actual addr iff root, use %pK : tunable via
       *   /proc/sys/kernel/kptr_restrict :
       *              = 0 : hashed addr [default]
       *              = 1 : _and_ root, actual addr displayed!
       *              = 2 : never displayed.
       *
       * DETAILS:
       * See https://www.kernel.org/doc/Documentation/printk-formats.txt
       */
      pr_info("kmalloc'ed memory dump (%d bytes @ %pK):\n", disp, kptr);
      mempattern(kptr, MY_PATTERN1, knum);
      print_hex_dump_bytes("", DUMP_PREFIX_ADDRESS, kptr, disp);

      vptr = vmalloc(42*PAGE_SIZE);
      if (!vptr) {
            pr_alert("vmalloc failed!\n");
            kfree(kptr);
            return -ENOMEM;
      }
      mempattern(vptr, MY_PATTERN2, PAGE_SIZE);
      pr_info("vmalloc'ed memory dump (%d bytes @ %pK):\n", disp, vptr);
      print_hex_dump_bytes("", DUMP_PREFIX_ADDRESS, vptr, disp);

      pr_info (
    "\nSome Kernel Details [sorted by decreasing address] -------------------\n"
#ifdef CONFIG_X86
      " FIXADDR_START       = 0x" FMTSPC "\n"
#endif
      " MODULES_END         = 0x" FMTSPC "\n"
      " MODULES_VADDR       = 0x" FMTSPC " [modules range: " FMTSPC_DEC " MB]\n"
#ifdef CONFIG_X86
      " CPU_ENTRY_AREA_BASE = 0x" FMTSPC "\n"
      " VMEMMAP_START       = 0x" FMTSPC "\n"
#endif
      " VMALLOC_END         = 0x" FMTSPC "\n"
      " VMALLOC_START       = 0x" FMTSPC " [vmalloc range: " FMTSPC_DEC " MB ="
FMTSPC_DEC " GB]" "\n"
      " PAGE_OFFSET         = 0x" FMTSPC " [lowmem region: start of all phy
mapped RAM (here to RAM-size)]\n",
#ifdef CONFIG_X86
            (TYPECST)FIXADDR_START,
#endif
            (TYPECST)MODULES_END, (TYPECST)MODULES_VADDR,
             (TYPECST)((MODULES_END-MODULES_VADDR)/(1024*1024)),
```

```
#ifdef CONFIG_X86
            (TYPECST)CPU_ENTRY_AREA_BASE,
            (TYPECST)VMEMMAP_START,
#endif
            (TYPECST)VMALLOC_END, (TYPECST)VMALLOC_START,
             (TYPECST)((VMALLOC_END-VMALLOC_START)/(1024*1024)),
             (TYPECST)((VMALLOC_END-VMALLOC_START)/(1024*1024*1024)),
            (TYPECST)PAGE_OFFSET);

#ifdef CONFIG_KASAN
      pr_info("\nKASAN_SHADOW_START = 0x" FMTSPC " KASAN_SHADOW_END = 0x" FMTSPC
"\n",
            (TYPECST)KASAN_SHADOW_START, (TYPECST)KASAN_SHADOW_END);
#endif
      /*
       * arch/x86/mm/dump_pagetables.c:address_markers[] array of structs would
       * be useful to dump, but it's private
      pr_info("address_markers = 0x" FMTSPC FMTSPC "\n", address_markers);
       */

      pr_info (
    "\nSome Process Details [sorted by decreasing address] ------------------\n"
      " [TASK_SIZE         = 0x" FMTSPC " size of userland]\n"
      " [Statistics wrt 'current' thread TGID=%d PID=%d name=%s]:\n"
      "        env_end     = 0x" FMTSPC "\n"
      "        env_start   = 0x" FMTSPC "\n"
      "        arg_end     = 0x" FMTSPC "\n"
      "        arg_start   = 0x" FMTSPC "\n"
      "        start_stack = 0x" FMTSPC "\n"
      "        curr brk    = 0x" FMTSPC "\n"
      "        start_brk   = 0x" FMTSPC "\n"
      "        end_data    = 0x" FMTSPC "\n"
      "        start_data  = 0x" FMTSPC "\n"
      "        end_code    = 0x" FMTSPC "\n"
      "        start_code  = 0x" FMTSPC "\n"
      " [# memory regions (VMAs) = %d]\n",
            (TYPECST)TASK_SIZE,
            current->tgid, current->pid, current->comm,
            (TYPECST)current->mm->env_end,
            (TYPECST)current->mm->env_start,
            (TYPECST)current->mm->arg_end,
            (TYPECST)current->mm->arg_start,
            (TYPECST)current->mm->start_stack,
            (TYPECST)current->mm->brk,
            (TYPECST)current->mm->start_brk,
            (TYPECST)current->mm->end_data,
            (TYPECST)current->mm->start_data,
            (TYPECST)current->mm->end_code,
            (TYPECST)current->mm->start_code,
            current->mm->map_count);

      pr_info (
      "\nSome sample kernel virtual addresses --------------------\n"
      "&statgul = 0x" FMTSPC ", &jiffies_64 = 0x%08lx, &vg = 0x" FMTSPC "\n"
      "kptr = 0x" FMTSPC " vptr = 0x" FMTSPC "\n",
```

```
            (TYPECST)&statgul, (long unsigned int)&jiffies_64, (TYPECST)&vg,
            (TYPECST)kptr, (TYPECST)vptr);

      foo();
      return 0;
}

static void __exit vm_img_exit(void)
{
      vfree (vptr);
      kfree (kptr);
      pr_info("Done.\n");
}

module_init(vm_img_init);
module_exit(vm_img_exit);

MODULE_AUTHOR("Kaiwan N Billimoria <kaiwan at kaiwantech dot com>");
MODULE_LICENSE("Dual GPL/MIT");

/*
 * -------------------------- Sample OUTPUT ----------------------------------
================================================================================
dmesg output of vm_show_kernel_seg.c on an x86_64 Linux system (Fedora 28)
```

```
[…]    << see output below >>


=========================================================================================
dmesg output of vm_show_kernel_seg.c on an ARM-32 Linux system (Qemu-emulated Vexpress)
=========================================================================================
vm_show_kernel_seg: loading out-of-tree module taints kernel.
vm_show_kernel_seg: module license 'Dual GPL/MIT' taints kernel.
Disabling lock debugging due to kernel taint
Platform:
 ARM-32 :
 32-bit OS
 Little-endian.
 sizeof(int)   =4, sizeof(long) =4
 sizeof(void *)=4, sizeof(u64 *)=4
kmalloc'ed memory dump (32 bytes @ 9ee94200):
9ee94200: ce fa ad de ce fa ad de ce fa ad de ce fa ad de  ................
9ee94210: ce fa ad de ce fa ad de ce fa ad de ce fa ad de  ................
vmalloc'ed memory dump (32 bytes @ a5346000):
a5346000: cc dd ee ff cc dd ee ff cc dd ee ff cc dd ee ff  ................
a5346010: cc dd ee ff cc dd ee ff cc dd ee ff cc dd ee ff  ................

Some Kernel Details [sorted by decreasing address] -------------------
 MODULES_END         = 0x80000000
 MODULES_VADDR       = 0x7f000000 [modules range: 00000016 MB]
 VMALLOC_END         = 0xff800000
 VMALLOC_START       = 0xa0800000 [vmalloc range: 00001520 MB =00000001 GB]
 PAGE_OFFSET         = 0x80000000 [lowmem region: start of all phy mapped RAM (here to RAM-
size)]

Some Process Details [sorted by decreasing address] ------------------
 [TASK_SIZE         = 0x7f000000 size of userland]
 [Statistics wrt 'current' thread TGID=736 PID=736 name=insmod]:
        env_end     = 0x7e992fef
        env_start   = 0x7e992f36
```

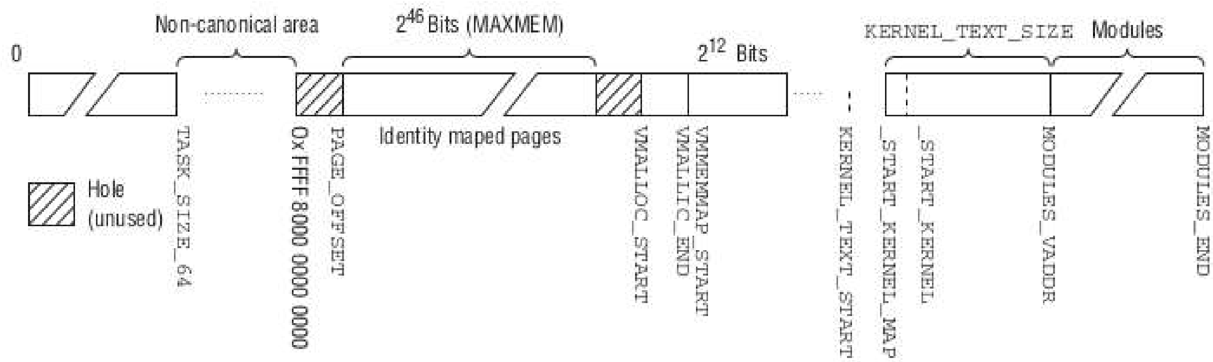
```
        arg_end     = 0x7e992f36
        arg_start   = 0x7e992f19
        start_stack = 0x7e992e20
        curr brk    = 0x000e4000
        start_brk   = 0x000c3000
        end_data    = 0x000c2719
        start_data  = 0x000c1f08
        end_code    = 0x000b1590
        start_code  = 0x00010000
 [# memory regions (VMAs) = 21]

Some sample kernel virtual addreses ---------------------
&statgul = 0x7f000908, &jiffies_64 = 0x80a02d00, &vg = 0x7f0006e4
kptr = 0x9ee94200 vptr = 0xa5346000
&c = 0x9ee89d87
---------------------Stack Dump:-------------------------------
CPU: 0 PID: 736 Comm: insmod Tainted: P           O    4.9.1 #4
Hardware name: ARM-Versatile Express
[<80110e98>] (unwind_backtrace) from [<8010cbc0>] (show_stack+0x20/0x24)
[<8010cbc0>] (show_stack) from [<803f5714>] (dump_stack+0xb0/0xdc)
[<803f5714>] (dump_stack) from [<7f000024>] (bar+0x24/0x34 [vm_show_kernel_seg])
[<7f000024>] (bar [vm_show_kernel_seg]) from [<7f002298>] (vm_img_init+0x298/0x2a8
[vm_show_kernel_seg])
[<7f002298>] (vm_img_init [vm_show_kernel_seg]) from [<80101d34>]
(do_one_initcall+0x64/0x1ac)
[<80101d34>] (do_one_initcall) from [<801ff884>] (do_init_module+0x74/0x1e4)
[<801ff884>] (do_init_module) from [<8019f188>] (load_module+0x1cc4/0x22f8)
[<8019f188>] (load_module) from [<8019f918>] (SyS_init_module+0x15c/0x17c)
[<8019f918>] (SyS_init_module) from [<80108220>] (ret_fast_syscall+0x0/0x1c)
---------------------------------------------------------------

===============================================================================
[OLDER] dmesg output of vm_img_lkm.c on an IA-32 Linux system
 (with indentation to make it human-readable)
===============================================================================
[79146.354770] vm_img_init:67 : 32-bit OS Little-endian.
[79146.354775] vm_img_init:77 : sizeof(int) = 4, sizeof(long) = 4 sizeof(void *)=4

[79146.354777] vm_img_init:85 : kmalloc'ed memory dump:
[79146.354780] efeae780: ce fa ad de ce fa ad de ce fa ad de ce fa ad de  ................
[79146.354782] efeae790: ce fa ad de ce fa ad de ce fa ad de ce fa ad de  ................

[79146.354795] vm_img_init:95 : vmalloc'ed memory start: 0xf95f6000

[79146.354802] vm_img_init:118 : Statistics wrt 'current'  [which right now is the
process/thread TGID 26512 PID 26512 name insmod]:
[79146.354802] PAGE_OFFSET = 0xc0000000 TASK_SIZE=0xc0000000
[79146.354802] VMALLOC_START = 0xf83fe000 VMALLOC_END=0xffbfe000 : vmalloc range:
0x00000078 MB
[79146.354802] MODULES_VADDR = 0xf83fe000 MODULES_END=0xffbfe000 : modules range:
0x00000078 MB
[79146.354802] start_code = 0xb7780000, end_code = 0xb77a56b0, start_data = 0xb77a6850,
end_data = 0xb77a71c0
[79146.354802] start_brk  = 0xb8e3f000, brk = 0xb8e60000, start_stack = 0xbf871260
[79146.354802] arg_start  = 0xbf8717aa, arg_end = 0xbf8717c1, env_start = 0xbf8717c1,
env_end = 0xbf871fef

[79146.354802] eg. kernel vaddr: &statgul = 0xfabf226c, &jiffies_64 = 0xc191e240, &vg =
0xfabf20c0
[79146.354802] kptr = 0xefeae780 vptr = 0xf95f6000
[79146.354809] foo:46 : &c = 0xf197fd6f

[79146.354810] bar:39 : ---------------------Stack Dump:-------------------------------
```

```
[79146.354813] CPU: 2 PID: 26512 Comm: insmod Tainted: P          OX 3.13.0-37-generic
#64-Ubuntu
[79146.354815] Hardware name: LENOVO 4291GG9/4291GG9, BIOS 8DET42WW (1.12 ) 04/01/2011
[79146.354816]  00000000 00000000 f197fd40 c1653867 ffbfe000 f197fd54 fabf0023 fabf1024
[79146.354822]  fabf1499 00000027 f197fd70 fabf00a7 fabf140f fabf1495 0000002e f197fd6f
[79146.354827]  7897fd7c f197fdfc fabf5348 fabf1150 fabf1489 00000076 00006790 00006790
[79146.354833] Call Trace:
[79146.354841]  [<c1653867>] dump_stack+0x41/0x52
[79146.354845]  [<fabf0023>] bar+0x23/0x80 [vm_img_lkm]
[79146.354848]  [<fabf00a7>] foo+0x27/0x4e [vm_img_lkm]
[79146.354851]  [<fabf5348>] vm_img_init+0x348/0x1000 [vm_img_lkm]
[79146.354859]  [<fabf5000>] ? 0xfabf4fff
[79146.354864]  [<c1002122>] do_one_initcall+0xd2/0x190
[79146.354867]  [<fabf5000>] ? 0xfabf4fff
[79146.354873]  [<c104c91f>] ? set_memory_nx+0x5f/0x70
[79146.354876]  [<c164f5f1>] ? set_section_ro_nx+0x54/0x59
[79146.354880]  [<c10c4371>] load_module+0x1121/0x18e0
[79146.354887]  [<c10c4c95>] SyS_finit_module+0x75/0xc0
[79146.354891]  [<c113a02b>] ? vm_mmap_pgoff+0x7b/0xa0
[79146.354901]  [<c1661bcd>] sysenter_do_call+0x12/0x12
[79146.354903] bar:41 : -----------------------------------------------------------

 */
$ make
...
$ sudo /bin/bash
... password:
#
# echo 1 > /proc/sys/kernel/kptr_restrict
#

$ sudo dmesg -C; sudo insmod ./vm_img_lkm.ko ; sudo dmesg
...

[85850.257391] Platform:
[85850.257395]  x86_64 :
[85850.257395]  64-bit OS
[85850.257397]  Little-endian.
[85850.257399]  sizeof(int)   =4, sizeof(long) =8
                sizeof(void *)=8, sizeof(u64 *)=8
[85850.257403] kmalloc'ed memory dump (32 bytes @ ffff90a789e13600):
[85850.257406] 0000000083c03d41: ce fa ad de ce fa ad de ce fa ad de ce fa ad
de  ................
[85850.257407] 000000008154b8a3: ce fa ad de ce fa ad de ce fa ad de ce fa ad
de  ................
[85850.257846] vmalloc'ed memory dump (32 bytes @ ffff9c540bc51000):
[85850.257849] 0000000097b70711: 88 99 aa bb cc dd ee ff 88 99 aa bb cc dd ee
ff  ................
[85850.257850] 0000000079d92e44: 88 99 aa bb cc dd ee ff 88 99 aa bb cc dd ee
ff  ................
[85850.257852]
               Some Kernel Details [sorted by decreasing address] -------------------
                FIXADDR_START       = 0xffffffffff576000
                MODULES_END         = 0xffffffffff000000
                MODULES_VADDR       = 0xffffffffc0000000 [modules range: 1008 MB]
                CPU_ENTRY_AREA_BASE = 0xfffffe0000000000
                VMEMMAP_START       = 0xffffeccec0000000
                VMALLOC_END         = 0xffffbc53ffffffff
                VMALLOC_START       = 0xffff9c5400000000 [vmalloc range: 33554431 MB =32767
GB]
```

```
                PAGE_OFFSET         = 0xffff90a3c0000000 [lowmem region: start of all phy
mapped RAM (here to RAM-size)]
[85850.257860]
               Some Process Details [sorted by decreasing address] ------------------
                [TASK_SIZE         = 0x00007ffffffff000 size of userland]
                [Statistics wrt 'current' thread TGID=20577 PID=20577 name=insmod]:
                       env_end     = 0x00007ffeb9904fe7
                       env_start   = 0x00007ffeb9904780
                       arg_end     = 0x00007ffeb9904780
                       arg_start   = 0x00007ffeb9904761
                       start_stack = 0x00007ffeb9903290
                       curr brk    = 0x000055a288079000
                       start_brk   = 0x000055a288058000
                       end_data    = 0x000055a287e63080
                       start_data  = 0x000055a287e61cf0
                       end_code    = 0x000055a287c610e7
                       start_code  = 0x000055a287c3e000
                [# memory regions (VMAs) = 35]
[85850.257867]
               Some sample kernel virtual addreses ---------------------
               &statgul = 0xffffffffc15f7410, &jiffies_64 = 0xffffffff88205000, &vg =
0xffffffffc15f7030
               kptr = 0xffff90a789e13600 vptr = 0xffff9c540bc51000
[85850.257870] &c = 0xffff9c540cf37c3f
[85850.257871] ---------------------Stack Dump:-------------------------------
[85850.257873] CPU: 2 PID: 20577 Comm: insmod Tainted: P           OE    4.16.13-
300.fc28.x86_64 #1
[85850.257874] Hardware name: LENOVO 20FMA089IG/20FMA089IG, BIOS R06ET39W (1.13 )
07/11/2016
[85850.257876] Call Trace:
[85850.257882]  dump_stack+0x5c/0x85
[85850.257887]  bar+0x16/0x22 [vm_show_kernel_seg]
[85850.257890]  foo+0x34/0x4e [vm_show_kernel_seg]
[85850.257893]  vm_img_init+0x30c/0x1000 [vm_show_kernel_seg]
[85850.257895]  ? 0xffffffffc133c000
[85850.257898]  do_one_initcall+0x48/0x13b
[85850.257902]  ? free_unref_page_commit+0x9b/0x110
[85850.257904]  ? _cond_resched+0x15/0x30
[85850.257907]  ? kmem_cache_alloc_trace+0x111/0x1c0
[85850.257910]  ? do_init_module+0x22/0x210
[85850.257912]  ? do_init_module+0x5a/0x210
[85850.257914]  ? load_module+0x210f/0x24a0
[85850.257917]  ? vfs_read+0x110/0x140
[85850.257920]  ? SYSC_finit_module+0xad/0x110
[85850.257922]  ? SYSC_finit_module+0xad/0x110
[85850.257925]  ? do_syscall_64+0x74/0x180
[85850.257927]  ? entry_SYSCALL_64_after_hwframe+0x3d/0xa2
[85850.257929] ---------------------------------------------------------------
```

Kernel Segment on the x86_64

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



Above diagram source: PLKA

From the above diagram, we can infer that:
maximum RAM supported to be directly-addressable (identity-mapped) by the Linux kernel on x86_64 = 2^46 (MAXMEM in above diagram) = 64 TB.

Also from *arch/x86/include/asm/pgtable_64_types.h*

```
...
/* See Documentation/x86/x86_64/mm.txt for a description of the memory map.
*/
#define MAXMEM          _AC(__AC(1, UL) << MAX_PHYSMEM_BITS, UL)
#define VMALLOC_START   _AC(0xffffc90000000000, UL)
#define VMALLOC_END     _AC(0xffffe8ffffffffff, UL)
#define VMEMMAP_START   _AC(0xffffea0000000000, UL)
#define MODULES_VADDR   (__START_KERNEL_map + KERNEL_IMAGE_SIZE)
#define MODULES_END     _AC(0xffffffffff000000, UL)
#define MODULES_LEN     (MODULES_END – MODULES_VADDR)
...
```

and from https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

```
Virtual memory map with 4 level page tables:

0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
                                      hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for
                                      hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys.
                                      memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
```

```
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB) kernel text mapping, from phys
0
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole

The direct mapping covers all memory in the system up to the highest
memory address (this means in some cases it can also include PCI memory
holes).

...
```

Kernel Segment on the Aarch64 (ARM-64)

See this kernel document.

Running a custom Ubuntu 18.04 Linux (kernel ver 4.15.0-1031-raspi2) compiled for 64-bit with 64-bit userspace root filesystem as well on the Raspberry Pi 3 Model B+ with the Aarch64 multicore processor [link]:

```
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[    0.000000] Linux version 4.15.0-1031-raspi2 (buildd@bos02-arm64-006) (gcc version
7.3.0 (Ubuntu/Linaro 7.3.0-16ubuntu3)) #33-Ubuntu SMP PREEMPT Wed Jan 16 09:52:45 UTC
2019 (Ubuntu 4.15.0-1031.33-raspi2 4.15.18)
[    0.000000] Machine model: Raspberry Pi 3 Model B Plus Rev 1.3

[ ... ]

[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffff000000000000 - 0xffff000008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffff000008000000 - 0xffff7dffbfff0000   (129022 GB)
[    0.000000]       .text : 0x         (ptrval) - 0x         (ptrval)   ( 11776 KB)
[    0.000000]     .rodata : 0x         (ptrval) - 0x         (ptrval)   (  4096 KB)
[    0.000000]       .init : 0x         (ptrval) - 0x         (ptrval)   (  6144 KB)
[    0.000000]       .data : 0x         (ptrval) - 0x         (ptrval)   (  1339 KB)
[    0.000000]        .bss : 0x         (ptrval) - 0x         (ptrval)   (  1035 KB)
[    0.000000]     fixed   : 0xffff7dfffe7fb000 - 0xffff7dfffec00000   (  4116 KB)
[    0.000000]     PCI I/O : 0xffff7dfffee00000 - 0xffff7dffffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffff7e0000000000 - 0xffff800000000000   (  2048 GB
maximum)
[    0.000000]               0xffff7faacc000000 - 0xffff7faacced0000   (    14 MB
actual)
[    0.000000]     memory  : 0xffffeab300000000 - 0xffffeab33b400000   (   948 MB)

[ ... ]
```

Notice how:

- the kernel text, rodata, etc addreses are *not displayed – security*
- the last line 'memory' is the *lowmem* region (phy RAM is mapped here).

Also:
Built a custom Aarch64-based Linux using the versatile Yocto project; then booted into it via it's runqemu script. See the (clipped) output below (guest RAM is 512 MB) :

```
$ runqemu qemuarm64 nographic

...
```

```
runqemu - INFO - Running <...>/qemu-system-aarch64 -netdev
tap,id=net0,ifname=tap0,script=no,downscript=no -device virtio-net-
device,netdev=net0,mac=52:54:00:12:34:02  -nographic -machine virt -cpu cortex-a57
-m 512 -drive id=disk0,file=<...>/tmp/deploy/images/qemuarm64/core-image-minimal-qemuarm64-
20161212132632.rootfs.ext4,if=none,format=raw -device virtio-blk-device,drive=disk0 -show-
cursor -device virtio-rng-pci -kernel <...>/tmp/deploy/images/qemuarm64/Image -append
'root=/dev/vda rw highres=off  console=ttyS0 mem=512M ip=192.168.7.2::192.168.7.1:255.255.255.0
console=ttyAMA0,38400'
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.8.3-yocto-standard (yocto@yocto-VirtualBox) (gcc version 6.2.0
(GCC) ) #1 SMP PREEMPT Mon Dec 12 21:03:15 IST 2016
[    0.000000] Boot CPU: AArch64 Processor [411fd070]
[    0.000000] Memory limited to 512MB

[...]

[    0.000000] Kernel command line: root=/dev/vda rw highres=off  console=ttyS0 mem=512M
ip=192.168.7.2::192.168.7.1:255.255.255.0 console=ttyAMA0,38400
[    0.000000] PID hash table entries: 2048 (order: 2, 16384 bytes)
[    0.000000] Dentry cache hash table entries: 65536 (order: 7, 524288 bytes)
[    0.000000] Inode-cache hash table entries: 32768 (order: 6, 262144 bytes)
[    0.000000] Memory: 497032K/524288K available (6396K kernel code, 758K rwdata, 1620K rodata,
640K init, 367K bss, 27256K reserved, 0K cma-reserved)
[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffffff8000000000 - 0xffffff8008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffffff8008000000 - 0xffffffbebfff0000   (   250 GB)
[    0.000000]       .text : 0xffffff8008080000 - 0xffffff80086c0000   (  6400 KB)
[    0.000000]     .rodata : 0xffffff80086c0000 - 0xffffff8008860000   (  1664 KB)
[    0.000000]       .init : 0xffffff8008860000 - 0xffffff8008900000   (   640 KB)
[    0.000000]       .data : 0xffffff8008900000 - 0xffffff80089bd800   (   758 KB)
[    0.000000]        .bss : 0xffffff80089bd800 - 0xffffff8008a1969c   (   368 KB)
[    0.000000]     fixed   : 0xffffffbefe7fd000 - 0xffffffbefec00000   (  4108 KB)
[    0.000000]     PCI I/O : 0xffffffbefee00000 - 0xffffffbeffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffffffbf00000000 - 0xffffffc000000000   (     4 GB  maximum)
[    0.000000]               0xffffffbf00000000 - 0xffffffbf00800000   (     8 MB actual)
[    0.000000]     memory  : 0xffffffc000000000 - 0xffffffc020000000   (   512 MB)
                                                          << 'lowmem' region – platform RAM >>
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1

[...]

Poky (Yocto Project Reference Distro) 2.2 qemuarm64 /dev/ttyAMA0

qemuarm64 login: root
root@qemuarm64:~# cat /proc/version
Linux version 4.8.3-yocto-standard (yocto@yocto-VirtualBox) (gcc version 6.2.0 (GCC) ) #1 SMP
PREEMPT Mon Dec 12 21:03:15 IST 2016
root@qemuarm64:~# cat /proc/cpuinfo
processor      : 0
BogoMIPS       : 125.00
Features       : fp asimd evtstrm aes pmull sha1 sha2 crc32
CPU implementer       : 0x41
CPU architecture: 8            << ARMv8 => 64-bit ARM CPU >>
```

```
CPU variant     : 0x1
CPU part        : 0xd07
CPU revision    : 0

root@qemuarm64:~#
...
```

Aarch64 kernel segment (On a Yocto Qemuarm64 platform running Linux 4.8.3)				
Kernel VA (top to bottom)	**In MB (base 10)**	**Size (MB)**	**Description**	
0xffffffc020000000	17592185782784			
0xffffffc000000000	17592185782272	512.00	Platform RAM (lowmem)	
4088 MB <gap>				
0xffffffbf00800000	17592185778184	8.00	vmemmap (actual)	
0xffffffbf00000000	17592185778176			
2 MB <gap>				
0xffffffbeffe00000	17592185778174		PCI I/O memory	
0xffffffbefee00000	17592185778158	16.00		
2 MB <gap>				
0xffffffbefec00000	17592185778156	4.00	Fixed	
0xffffffbefe7fd000	17592185778152			
1000.099609 MB <gap>				
0xffffffbebfff0000	17592185777152		Vmalloc (250 GB) . . .	
256885.8008	*vmalloc*			
0xffffff8008a1969c	17592185520266	0.40	BSS (368 Kb)	Kernel (zImage) Mapping
0xffffff80089bd800	17592185520266	0.70	Data (758 Kb)	
0xffffff8008900000	17592185520265	0.60	Init (640 Kb)	
0xffffff8008860000	17592185520264	1.60	Roinit (1664 Kb)	
0xffffff80086c0000	17592185520263	6.30	Text (6400 Kb)	
0xffffff8008080000	17592185520257			
0xffffff8008000000	17592185520256			
0xffffff8000000000	17592185520128	128.00	Kernel Modules	

Yet another example – Buildroot's Aarch64 Qemu system's

See <buildroot>/board/qemu/aarch64-virt/readme.txt

. . .

```
Linux version 4.1.0 (kaiwan@kaiwan-T460) (gcc version 5.4.0 (Buildroot
2016.11.2) ) #4 SMP PREEMPT Fri Jan 27 11:52:44 IST 2017
CPU: AArch64 Processor [411fd070] revision 0
Detected PIPT I-cache on CPU0

[...]

Memory: 415648K/524288K available (5587K kernel code, 440K rwdata, 2224K
rodata, 440K init, 194K bss, 92256K reserved, 16384K cma-reserved)
Virtual kernel memory layout:
    vmalloc : 0xffffff8000000000 - 0xffffffbdbfff0000   (   246 GB)
    vmemmap : 0xffffffbdc0000000 - 0xffffffbfc0000000   (     8 GB maximum)
              0xffffffbdc1000000 - 0xffffffbdc1800000   (     8 MB actual)
    fixed   : 0xffffffbffabfd000 - 0xffffffbffac00000   (    12 KB)
    PCI I/O : 0xffffffbffae00000 - 0xffffffbffbe00000   (    16 MB)
    modules : 0xffffffbffc000000 - 0xffffffc000000000   (    64 MB)
    memory  : 0xffffffc000000000 - 0xffffffc020000000   (   512 MB)
      .init : 0xffffffc000823000 - 0xffffffc000891000   (   440 KB)
      .text : 0xffffffc000080000 - 0xffffffc000822494   (  7818 KB)
      .data : 0xffffffc000897000 - 0xffffffc000905000   (   440 KB)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
[…]
```

Kernel Paging Tables

There is just 1 set of paging tables per process, even when the process switches to kernel mode. The kernel PTEs are mapped and the same for all usermode processes. (This is why *mm* and *active_mm* tend to be the same value).

However, when kernel code does a vmalloc() it's setting up a virtual region; which thus faults upon first access and raises a page fault; the kernel "fixes" the fault by by copying the kernel master page table entries (pte's) that changed into that of 'current' (syncing the tables to that of the kernel)!

Brief Note on Kernel Paging Table Initialization
[Source]

… Thereafter, *setup_arch()* calls *paging_init()* in *arch/i386/mm/init.c*. This function does several things. First, it calls *pagetable_init()* to map the entire physical memory, or as much of it as will fit between PAGE_OFFSET and 4GB, starting at PAGE_OFFSET.

In *pagetable_init()*, we actually build the kernel page tables in swapper_pg_dir that maps the entire physical memory range to PAGE_OFFSET. This is simply a matter of doing the arithmetic and stuffing the correct values into the page directory and page tables.

This mapping is created in *swapper_pg_dir*, the kernel page directory; this is also the page directory used to initiate paging.

```
<<
mm/init.c
struct mm_struct init_mm = {
    .mm_rb        = RB_ROOT,
    .pgd          = swapper_pg_dir,
    .mm_users     = ATOMIC_INIT(2),
    .mm_count     = ATOMIC_INIT(1),
    .mmap_sem     = __RWSEM_INITIALIZER(init_mm.mmap_sem),
    .page_table_lock =  __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
    .mmlist       = LIST_HEAD_INIT(init_mm.mmlist),
    INIT_MM_CONTEXT(init_mm)
};
>>
```

… If there is physical memory left unmapped here - that is, memory with physical address greater than 4GB-PAGE_OFFSET - that memory is unusable unless the CONFIG_HIGHMEM option is set. …

SIDEBAR
Regarding kernel paging, a question your author asked on StackOverflow and the "correct" answer; please read: **How exactly do kernel virtual addresses get translated to physical RAM?**

mem_map:

All physical memory pages (page frames) are tracked via the *mem_map[]* array:

```
include/linux/mmzone.h:
extern struct page *mem_map;
```

Also, given a kernel virtual address, one can convert to it's corresponding page structure by:

```
#define virt_to_page(kaddr) pfn_to_page(__pa(kaddr) >> PAGE_SHIFT)
```

Explanation:

\>> PAGE_SHIFT = >> 12 (on 4K page size) = divide by 2^12 (4096)

So, convert the kernel va to a physical address, and divide it by PAGE_SIZE; this gives us the PFN - page frame number (in effect, the index into the mem_map array!).

Convert to struct page * via the pfn_to_page() API.
 	
[FYI, there are three (physical) memory models:
(code shown below is relevant to x86 architecture):
 	
• for the CONFIG_FLATMEM case: *<< typically embedded systems >>*
`#define __pfn_to_page(pfn)  (mem_map + ((pfn) - ARCH_PFN_OFFSET))`
 	
• for the CONFIG_DISCONTIGMEM case: *<< older >>*
`#define __pfn_to_page(pfn)          \`
`        ({  unsigned long __pfn = (pfn);        \`
`             unsigned long __nid = arch_pfn_to_nid(__pfn);  \`
`             NODE_DATA(__nid)->node_mem_map + arch_local_page_offset(__pfn,`
`__nid);\`
`        })`
 	
• for the CONFIG_SPARSEMEM_VMEMMAP case: *<< x86_64 default [N]UMA >>*
`#define vmemmap ((struct page *)VMEMMAP_START)`
`...`
`/* memmap is virtually contiguous.  */`
`#define __pfn_to_page(pfn)  (vmemmap + (pfn))`
]

<< End of Linux Memory Management, Part 3 >>

kaiwa
Please *on returns,* *techni*