Memory Subsystem and Data Types in the Linux Kernel

Praktikum Kernel Programming

Björn Brömstrup and Alexander Koglin

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

21. January 2015

Outline

Contents

Memory Addresses in the Kernel

- Physical Addresses
- (Kernel) Logical Addresses
  - normal Kernel address space
  - 1-to-1 mapping to physical memory
    - subtract `PAGE_OFFSET` (0xC000000 on 32 bits $\Rightarrow$ 3:1 split)
  - uses hardware's native pointer size $\Rightarrow$ with 32 bits probably not all memory can be logically addressed (max. 896 MB)
  - Mapping by Memory Management Unit (MMU) between CPU and memory bus
- (Kernel) Virtual Addresses
  - also mapping from kernel space address to physical address
  - not necessarily 1-to-1 mapping
  - able to allocate physical memory that has no logical address
  - Limited addresses ranges reserved: vmalloc is 128 MB

Memory Zones

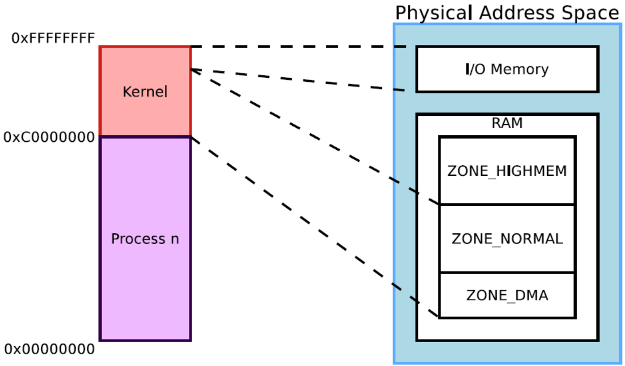

Figure: `http://free-electrons.com/doc/training/linux-kernel/linux-kernel-slides.pdf`

Memory Zones

Physical memory is divided into three zones:

- `ZONE_DMA`
  - capable of DMA (Direct Memory Access)
- `ZONE_NORMAL`
  - normal Kernel memory
- `ZONE_HIGHMEM`
  - not mapped by kernel

High Memory

- Low memory
  - Memory for which logical addresses exist in kernel space
- High Memory
  - `PAGE_OFFSET` 0xC000000 on 32 bits $\Rightarrow$ only 1 GB for Kernel addressable
  - not directly addressable part is called High Memory
  - temporary mapping: `kmap` to insert memory page into page table
  - `kunmap` to eject

Virtual Memory

- independent extension of the physical space
- logical addresses are part of the virtual address space

Features:

- **Large Address Spaces:** OS appears as if it has a larger amount of memory than it actually has + Swapping
- **Fair Physical Memory Allocation**
- **Memory Mapping** maps directly into the virtual address space of a process
- **Security:** Each process has own seperate virtual address space
- **Shared Virtual Memory** allows to share memory (e.g code) between processes

Pages

- Physical and Virtual Memory divided into chunks of the same size called pages (4 KB on x86)
- use of page tables for translation
- easier translation
- each page has unique page frame number (PFN)
- an address consists of offset and (virtual) PFN $\Rightarrow$ look up (physical) PFN and access at correct offset
- translation lookaside buffer (TLB)
- flags indicate if the page is in real memory
- Swapping/Paging

Memmap

```
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 ...
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
```

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

Contents

Kernel Memory Allocators

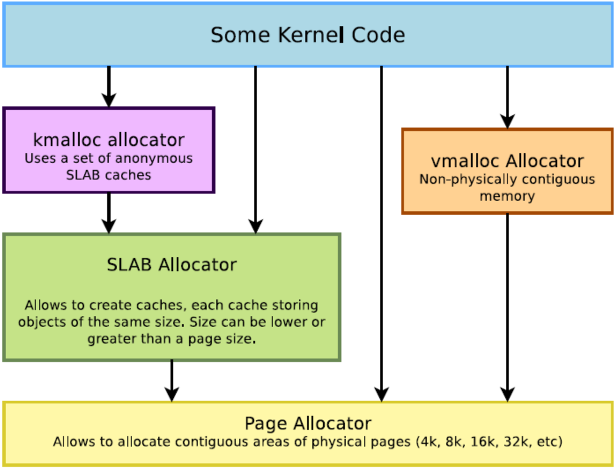

Figure: `http://free-electrons.com/doc/training/linux-kernel/linux-kernel-slides.pdf`, 252

Page Allocator

- gets a power of two of `PAGE_SIZE` of physically contiguous memory (Buddy Allocator)
- page size on x86 is 4 KB
- size up to about 8 MB (medium size)
- physical memory fragmentation $\Rightarrow$ limited size
  - fails when `order`$= \log_2(\textit{number\_of\_pages})$ too big
  - see `/proc/buddyinfo` for info about the memory zones' available blocks of each order

Page Allocator API

- `unsigned long __get_free_page(int flags)`
  - returns (virtual) address of a free page
  - `flags`
- `unsigned long get_zeroed_page(int flags)`
- `unsigned long __get_free_pages(int flags, unsigned int order)`
  - returns (virtual) address of beginning of memory area consisting of multiple contiguous pages
  - `order`$= \log_2(\textit{number\_of\_pages})$
    - `number_of_pages` must be of power two
    - `get_order(number_of_pages)`

Page Allocator Flags

- `GFP_KERNEL`
  - primary flag for memory allocation
  - blocking
- `GFP_ATOMIC`
  - non-blocking
  - for critical sections
  - can fail
- `GFP_DMA`
  - allocator for DMA suitable memory
- more available under `include/linux/gfp.h`

Page Allocator API

- `void free_page(unsigned long addr)`
- `void free_pages(unsigned long addr,unsigned int order)`
  - multiple pages
  - same order as in allocation is imperative!

SLAB Allocator

- creates caches containing objects of the same size
- relies on the page allocator
- object size can nonetheless be bigger than acual page size
- dynamic cache size management (info `/proc/slabinfo`)
- API: `include/linux/slab.h`

- usecase: inherently by the kernel for data structures
  - file objects
  - directory entries
  - network package descriptors
  - ...
- but rarely by drivers

Different Implementations of SLAB

- three different, but API compatible
  - SLAB: legacy
  - SLUB (Unqueued Allocator): default, simpler, scales well for huge systems, less fragmentation
  - SLOB (Simple List Of Blocks): simple, space efficient, but poor scalability, used for embedded systems

kmalloc Allocator

- main kernel memory allocator (since 1.0 available)
- allocates physically contiguous buffers
- although only mandatory for hardware devices it's faster than `vmalloc`
- case analysis:
  - small size: uses SLAB caches (`kmalloc-XXX` in `/proc/slabinfo`)
  - larger size: uses page allocator
- size (x86):
  - at least as much as you ask for
  - typical at least 32 bytes
  - max per allocation: 4 MB (assuming 4 KB page size), but typical 128 KB
  - total: 128 MB

kmalloc

- include <linux/slab.h>
- get a (virtual memory) pointer to a buffer:
- `void *kmalloc(size_t size, int flags);`
  - size: number of bytes to allocate
  - flags: same as for page allocator
    - GFP_KERNEL
    - GFP_ATOMIC
    - GFP_DMA
- `void kfree(const void *objp);`

kmalloc related functions

zero-initialized variations:

- `void *kzalloc(size_t size, gfp_t flags);`
  - $\neq$ `zalloc`
  - since 2.6.14
- `void *kcalloc(size_t n, size_t size, gfp_t flags);`
  - like `calloc`

reallocation:

- `void *krealloc(const void *p, size_t new_size, gfp_t flags);`
  - like `realloc`: changes size of buffer pointed to by `p` to `new_size`.

kmalloc: Managed Ressources:

Motivation: Don't initialize a PCI/USB on `module_init`!
Solution: On device activation the kernel automatically selects the device's name/ID matching driver and calls its `probe`-function.

- `void *devm_kmalloc(struct device *dev, size_t size, int flags);`
- `*devm_kzalloc, *devm_kcalloc`
- 2.6.21: Managed Ressources

⇒ auto-free allocated buffers when the device or module is detached or an error occurs (in the initialization).
⇒ less errors/memory leaks

vmalloc Allocator

- implemented in Linux/mm/vmalloc.c
- declared in include/linux/vmalloc.h
  - `void *vmalloc(unsigned long size);`
    - at least size bytes (rounded to the next page)
  - `void vfree(void *addr);`
- allocated memory is only virtually contiguous
- allocates noncontiguous chunks of physical memory and maps it via page tables into a contiguous chunk of the virtual address space (prefers `ZONE_HIGHMEM`)
- large allocations possible (but on 32 bits only 128 MB in total)
- slower than `kmalloc`
- not usable for DMA (exeption: SPARC with DVMA)

Large Buffers: Bootmem

- physical memory fragmentation
- large contiguous buffer allocations could fail
- Do you need it? Really? Alterantive: Scatter and Gather?!
- Solution: Allocate memory at boot time $\Rightarrow$ bypass limitations
- private pool
- freed memory possibly not reuseable
- only Kernel drivers directly linked to the kernel
- `include <linux/bootmem.h>`
  - `void *alloc_bootmem_pages(unsigned long size);`
  - `void *alloc_bootmem_low_pages(unsigned long size);`

Large Buffers: CMA

- contiguous memory allocation (CMA)
- grabs a chunk of contiguous physical memory at boot time
- drivers can request memory
- areas `cma=v=20M,c=20M cma_map=video=v;camera=c`
- `include <linux/cma.h>`
  - `unsigned long cma_alloc(const struct device *dev, const char *kind, unsigned long size, unsigned long alignment);`

Debugger

- Kmemcheck
  - Dynamic checker for access to uninitialized memory.
  - works best on x86
- Kmemleak Dynamic checker for memory leaks

Contents

kfifo

<linux/kfifo.h>: A simple data queue

Enqueue:

- `kfifo_in(kfifop, datap, length)`

Dequeue:

- `kfifo_out(kfifop, datap, length)`

Peek:

- `kfifo_out_peek(kfifop, datap, length, offset)`

Clear:

- `kfifo_reset(kfifop)`

kfifo

Creating and destroying a kfifo

```
struct kfifo queue;
int err;
err = kfifo_alloc(&queue, QUEUE_SIZE, GFP_FLAGS);
// ...
void kfifo_free(&queue);
```

Other useful functions

- `kfifo_size()`, `kfifo_len()`, `kfifo_avail()`
- `kfifo_from_user()`, `kfifo_to_user()`
- `kfifo_dma_...()`

list

<linux/list.h>: A doubly-linked, circular, **intrusive** list

- ```
  struct list_head {
      struct list_head *next, *prev;
  };
  ```
- ```
  #define list_entry(nodep, contnr_t, member_name) \
      container_of(nodep, contnr_t, member_name)
  ```
- ```
  void INIT_LIST_HEAD(struct list_head *list) {
      list->next = list;
      list->prev = list;
  }
  ```
  ```

Manipulating a list

Adding and deleting elements

- `list_add(newp, nodep)`
- `list_add_tail(newp, nodep)`
- `list_del(nodep)`
- `list_replace(oldp, newp)`

Other useful functions

- Rotating
- Cutting and splicing
- Moving entries from one list to another
- Sorting (in <linux/list_sort.h>)

Iterating over a list

Over list_heads:

- `nodep = nodep->next;`
- `nodep = nodep->prev;`
- `list_for_each(nodep, headp) { /* use nodep */ }`
- `list_for_each_safe(nodep, nextp, headp) { /* ... */ }`

Over list entries:

- `objp = list_next_entry(objp, member_name)`
- `objp = list_prev_entry(objp, member_name)`
- ```
  list_for_each_entry(objp, headp, member_name) {
      /* use objp */
  }
  ```
  ```

list example

Example: list_demo.c

Other list implementations

Other list implementations

- <linux/klist.h>: A wrapper around list_head for thread safe access and modification
- <linux/llist.h>: A singly-linked, lock-less list
- <linux/plist.h>: A priority list

rbtree

<linux/rbtree.h>: An intrusive red-black tree

- ```
  struct rb_root {
      struct rb_node *rb_node;
  };
  ```
- ```
  struct rb_node {
      unsigned long __rb_parent_color;
      struct rb_node *rb_left;
      struct rb_node *rb_right;
  }
  ```
- ```
  #define rb_entry(nodep, contnr_t, member_name) \
      container_of(nodep, contnr_t, member_name)
  ```
  ```

Using an rbtree

There is no predefined function to search an rbtree. You can walk through the tree using these methods:

- `nodep = nodep->rb_left;`
- `nodep = nodep->rb_right;`
- `nodep = rb_parent(nodep);`
- `rb_next(), rb_prev(), rb_first(), rb_last()`

Inserting a node is a two step process

- `rb_link_node(nodep, parentp, &parentp->rb_left);`
  `rb_insert_color(nodep, rootp);`

Deleting a node

- `rb_erase(nodep, rootp);`

rbtree example

Example: rbtree_demo.c

(rbtree is also very well documented in Documentation/rbtree.txt)

Other tree implementations

Other tree implementations

- <linux/btree.h>: A B+Tree
- <linux/radix-tree.h>: Maps integers to pointers

hashtable

<linux/hashtable.h>: An intrusive hashtable

A hashtable is an array (!) of hlist_heads

- ```
  #define DEFINE_HASHTABLE(name, bits)                \
      struct hlist_head name[1 << (bits)] = {         \
          [0 ... ((1<<(bits))-1)] = HLIST_HEAD_INIT   \
      }
  ```
- ```
  #define hash_add(tbl, node, key)                    \
      hlist_add_head(node,                            \
                     &tbl[hash(key,ARRAY_SIZE(tbl))])
  ```

hashtable example

```
struct obj {
    struct data data;
    struct hlist_node hash_node;
    int id;
};
static DEFINE_HASHTABLE(tbl, 8); //tbl has 256 buckets

struct obj *swap_out(struct obj *in, int out_id) {
    struct obj *obj;
    int i;
    hash_add(tbl, &in->hash_node, in->id);
    hash_for_each_possible(tbl, obj, hash_node, out_id) {
        if(obj->id == out_id) {
            hash_del(tbl, obj->hash_node);
            return obj;
        }
    }
}
```

idr

<linux/idr.h>: Maps unique ids to pointers

Initializing and destroying an idr

- ```
  struct idr id_map;
  idr_init(&id_map);
  ```
- ```
  idr_destroy(&id_map);
  ```

Allocating, finding and removing unique ids

- ```
  int uid, err;
  do {
      if(!idr_pre_get(&id_map, GFP_FLAGS))
          return -ENOSPC;
      err = idr_get_new(&id_map, ptr, &uid);
  while(err  == -EAGAIN);
  ```
- ```
  ptr = idr_find(&id_map, uid);
  ```
- ```
  idr_remove(&id_map, uid);
  ```
  ```

Contents

Summary

Allocator	Property		
	physically contiguous	typ. size	max size
Page Allocator	Yes		8 MB
SLAB Allocator	No		
kmalloc	Yes	128 KB	4 MB
vmalloc	No	arbitrary	128 MB on 32bits
Large Buffers	Yes		

- include `<linux/slab.h>`
- `void *kmalloc(size_t size, int flags);`
- flags: `GFP_KERNEL`, `GFP_ATOMIC`, `GFP_DMA`
- `void kfree(const void *objp);`

- declared in `<linux/vmalloc.h>`
- `void *vmalloc(unsigned long size);`
- `void vfree(void *addr);`

Summary

- The Linux kernel has generic implementations of the most used data structures
- They are implemented in `lib/`
- Look through the header files, and if you are unsure about something, the implementation.

- `<linux/kfifo.h>`:
- `<linux/list.h>`:
- `<linux/rbtree.h>`
- `<linux/hashtable.h>`

Thank you for listening.

Questions?

Contents

Literature

Jonathan Corbet.
idr - integer id management.
2004.

Jonathan Corbet.
Contiguous memory allocation for drivers.
2010.

Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman.
Linux Device Drivers, 3rd Edition.
O'Reilly Media, Inc., 2005.

Jake Edge.
A generic hash table.
2012.

Linux Kernel and Driver Development Training.
Free Electrons, 2015.

Robert Love.
Linux Kernel Development.
Developer's library : essential references for programming professionals.
Addison-Wesley, 2010.