Low-level memory management

Memblock

A primitive memory manager

Used at boot time

- Until higher-level code takes over

For details

- See .../include/linux/memblock.h

- (Memblock was formerly called LMB)

Pages

The fundamental unit of memory management

- 4096 bytes on most systems

Most hardware can do multiple sizes

- Linux support is limited

struct page

in .../include/linux/mm_types.h

```
struct page {
    unsigned long flags;
    atomic_t count;
    unsigned long private;
    struct address_space *mapping;
    struct list_head lru;
    void *virtual;
    /* ... */
}
```

The system memory map

One struct page for every real page

Layout can be complicated

...if memory layout is complicated

Zones

Divide pages by hardware features

- DMA reachability
- NUMA node

Most allocations specify zone info

- ...at least implicitly

Details:

- `struct zone` in <linux.mmzone.h>

Typical zones

ZONE_DMA
- Addressable with 24 bits

ZONE_DMA32
- Addressable with 32 bits

ZONE_NORMAL
- Most memory

ZONE_HIGHMEM
- Not directly addressable

High memory

32-bit systems can address 4GB

- User space gets 3GB
- Kernel code takes some space
- -> About 900MB directly addressable

The rest is “high memory”

- Lives in ZONE_HIGHMEM
- Not directly addressable from the kernel

Accessing high memory

From <linux/highmem.h>

```
void *kmap(struct page *page);
void kunmap(struct page *page);

void *kmap_atomic(struct page *page);
void kunmap_atomic(struct page *page);
```

High memory from user space

“just works”

Page allocation concepts

- Order
  - Allocations are done in powers of two (e.g. 1, 2, 4, or 8 pages)
  - The “order” = log2(npages)
    - An “order 2” allocation is 4 pages
- Higher-order allocation reliability
  - 0 - “always works”
  - 1 - usually works
  - higher - more uncertain

The buddy allocator

The page allocator maintains groups of power-of-two sizes

See /proc/buddyinfo

```
zone    DMA      1    1   2   2    2   3   3   1   2   2   0
zone Normal  5364 4611 477  12    0   0   0   0   0   0   1
zone HighMem 2011 1113 342  60    3   0   0   0   0   0   0
```

Page allocation concepts

GFP flags

- Describe how the allocation is to happen

These flags control

- Whether the operation can block
- Which zone(s) can be used
- What can be done to obtain memory
- ...

GFP flags

GFP_KERNEL

- Normal kernel allocation
- Can block
- No high memory

GFP_ATOMIC

- Atomic kernel allocation
- Will not block
- No high memory
- More likely to fail

GFP_FLAGS

GFP_NOFS
- Like GFP_KERNEL
- Do not call into filesystem code

GFP_NOIO
- GFP_KERNEL + do not start I/O

GFP_USER, GFP_HIGHUSER
- Pages for user-space use
- GFP_HIGHUSER can use highmem

Basic allocation

Use one of:

```
struct page *alloc_page(gfp_t mask);
unsigned long get_free_page(gfp_t mask);
```

Less-basic allocation

```
struct page *alloc_pages(gfp_t mask, int order);
struct page *alloc_pages_node(int node_id,
                         gfp_t mask, int order);

/* Several others */

void *alloc_pages_exact(size_t size, gfp_t mask);

unsigned long get_free_pages(gfp_t mask, int order);
unsigned long get_zeroed_page(gfp_t gfp_mask);
   /* can also use __GFP_ZERO */
```

Page allocator tracepoints

mm_page_alloc
mm_page_alloc_extfrag
mm_page_free_direct

Slab allocators

An allocator for small objects

- (kernel data structures)
- Built on the page allocator

The kernel has three alternatives

- Slab: the classic allocator, still fastest?
- SLUB: alternative optimized allocator
- SLOB: space-efficient allocator

Slab creation

```
struct kmem_cache *kmem_cache_create(
    const char *name,
    size_t object_size,
    size_t alignment,
    unsigned long flags,
    void (*constructor)(void *));

struct kmem_cache *KMEM_CACHE(struct, flags);

void kmem_cache_destroy(struct kmem_cache *c);
    /* All objects must be free first */
```

Object allocation

```
void *kmem_cache_alloc(struct kmem_cache *c
                       gfp_t flags);
void kmem_cache_free(struct kmem_cache *c,
                     void *object);

/* Or also... */
void *kmalloc(size_t size, gfp_t flags);
void *kmalloc_node(size_t size, gfp_t flags,
                   int node_id);
void *kzalloc(size_t size, gfp_t flags);

void kfree(void *object);
```

Slab tracepoints

kmem_cache_alloc
kmem_cache_alloc_node
kmem_cache_free
kmalloc
kfree

See also:

- /proc/slabinfo

vmalloc()

Allocates virtually contiguous space
- (in kernel space)
- Physically scattered memory
- Expensive to use

```
void *vmalloc(unsigned long size);
void *vzalloc(unsigned long size);
void *vmalloc_user(unsigned long size);
/* ... */
void vfree(void *addr);
```

mempools

When memory allocations cannot fail

```
#include <linux/mempool.h>

mempool_t *mempool_create(int min_objects,
                          mempool_alloc_t alloc_fn,
                          mempool_free_t free_fn,
                          void *pool_data);
void *mempool_alloc(mempool_t *pool, gfp_t mask);
void mempool_free(void *obj, mempool_t *pool)
```

Accessing I/O memory

- Device memory must be accessed specially
  - Especially for portable code

ioremap()

To make device memory available:

```
void __iomem *ioremap(phys_addr_t addr,
                      unsigned long size);
void __iomem *ioremap_nocache(phys_addr_t addr,
                      unsigned long size);
```

Returns the address of the mapping

...sort of

Accessing I/O memory

Use:

```
u8 readb(void __iomem *addr);
u16 readw(void __iomem *addr);
u32 readl(void __iomem *addr);
void writeb(u8 v, void __iomem *addr);
void writew(u16 v, void __iomem *addr);
void writel(u32 v, void __iomem *addr);
/* memcpy/memset equivalents too */
```

Low-level memory management questions?