Linux Kernel Programming

Redha Gouicem and Julien Sopena

Lecture 05:

Memory Management

Memory Organization

Pages are the basic unit of memory management.

Even if memory is byte/word addressable, the smallest *management unit* is the page, to accommodate fast lookups and address translations.

Some definitions:

A **page** (or *virtual page*) is a fixed-size block of contiguous **virtual** memory.

A **page frame** (or *physical page*) is a fixed-size block of contiguous **physical** memory.

Page size depends on the architecture, some of them even support multiple sizes.

Supported page sizes (non-exhaustive)

Architecture	4 KiB	16 KiB	64 KiB	2 MiB	4 MiB	32 MiB	512 MiB	1 GiB
x86_64	X			X				X
armv7	X		X					
aarch64	X	X	X	X		X	X	X
riscv32	X				X			
riscv64	X			X				X

Kernel Representation of Pages

The kernel maintains a `struct page` for each **page frame** available on the system.
The structure is defined in `include/linux/mm_types.h`.

Here is a simplified definition of the structure:

```c
struct page {
    unsigned long           flags;      // page status, flags available in include/linux/page-flags.h
    atomic_t                _refcount;  // number of references to this frame

    /* page cache and anonymous pages */
    atomic_t                _mapcount;  // number of page tables this frame is mapped in
    struct address_space    *mapping;   // if used in page cache, object associated to this frame
    struct list_head        lru;        // least-recently used list for eviction

    void                    *virtual;   // kernel virtual address when not kmapped, used when using high memory
};
```

Since there is one `struct page` per **physical page**, isn't this a lot of memory for metadata?

In practice, a `struct page` is only around 40 bytes (lots of unions in there).

So, in a system with 16 GiB of memory and 4KiB pages, you will have: $\frac{17{,}179{,}869{,}184}{4{,}096} = 4,194,304$ pages

Which means *only* ~160 MiB, or less than 10% of the total memory.

You can reduce this metadata footprint by increasing the page size.

Zones

Not all addresses are equal in hardware, so all frames are not treated identically.
The kernel separates pages in multiple **zones** with different properties.

The two main hardware limitations that require zones are:

- Some hardware can only do Direct Memory Accesses (DMA) to certain addresses;
- Some architectures have a physical address space larger than their virtual address space, which means that some frames are not permanently mapped into the kernel address space.

Linux separates the physical memory into four main zones:

- `ZONE_DMA` contains frames that can be accessed through DMA;
- `ZONE_DMA32` is the same as `ZONE_DMA`, but only for 32-bit devices;
- `ZONE_NORMAL` contains regularly mapped pages (DMA is also possible);
- `ZONE_HIGHMEM` contains pages that have physical addresses bigger than the virtual address space allows.

Zones on x86-32 (from *Linux Kernel Development, 3rd Edition, Robert Love*)

Zone	Description	Physical Memory
`ZONE_DMA`	DMA-able pages	$<$ 16 MiB
`ZONE_NORMAL`	Normally addressable pages	16–896 MiB
`ZONE_HIGHMEM`	Dynamically mapped pages	$>$ 896 MiB

Memory API

Quick recap of the memory management API in the kernel:

- Allocate pages

```
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order); // allocate 2^order contiguous physical pages, return the first page
void *page_address(struct page *page); // get the actual address of the page from it's struct page
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order); // same as alloc_pages(), but returns the address directly
struct page *alloc_page(gfp_t gfp_mask);   // wrapper to allocate a single page
unsigned long __get_free_page(gfp_t gfp_mask); // wrapper to allocate a single page and get the actual address
unsigned long get_zeroed_page(gfp_t gfp_mask); //same as __get_free_page(), but the page is filled with zeros
```

- Free pages

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

> **Important**
>
> Careful to not free pages **you** did not allocate. That would most likely break the system.

- Allocate arbitrary sizes

```
void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *ptr);
void *vmalloc(unsigned long size);
void vfree(const void *addr);
```

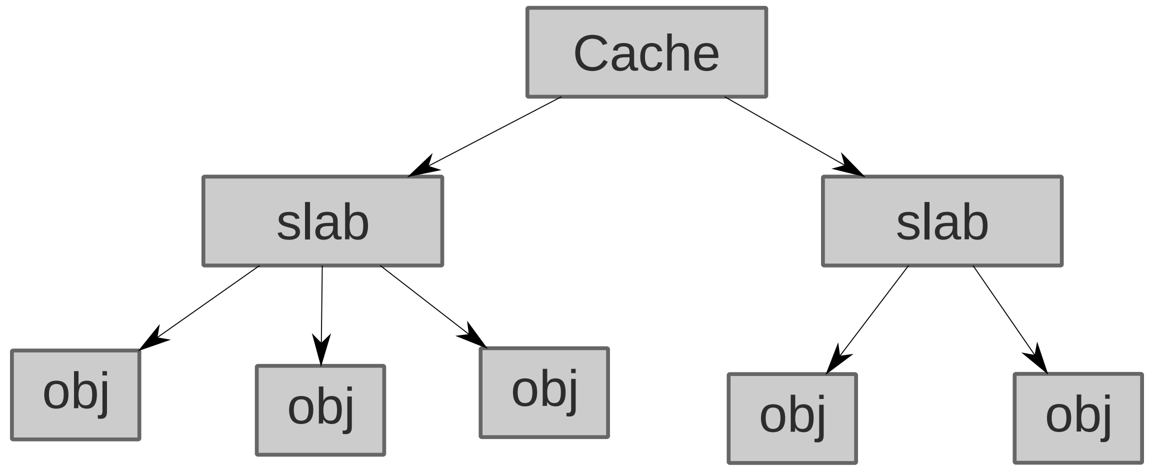
The Slab Layer

Allocating and freeing objects is extremely frequent, so it's a good idea to have some sort of caching mechanism.

In Linux, that caching mechanism is called the slab layer.

The slab layer allows you to create **caches**, each of which contain a certain type of objects, *e.g.*, `struct task_struct` or `struct inode`.

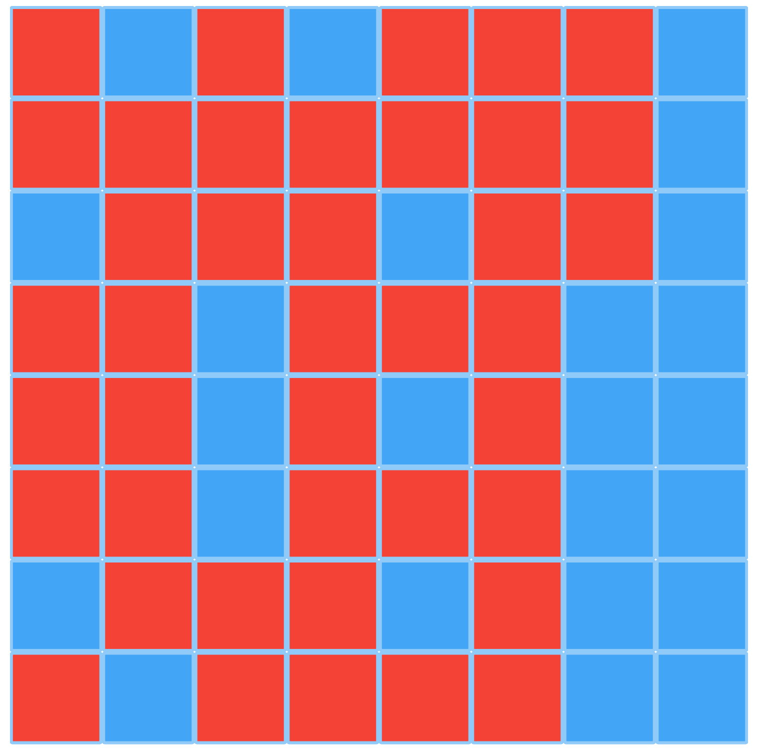
Each cache is then divided into **slabs**, blocks of contiguous memory that contain a certain number of instances of the object stored by this cache.



A **slab** contains the actual data and maintains their status (used or free).

When a slab is full, the slab layer will allocate a new one for this cache.

When the system wants to reclaim memory, empty slabs will be freed.



> **Note**
>
> Additionally, allocations are done at the page granularity, so for smaller objects, you would need manual management to not waste memory.

SLAB Allocator

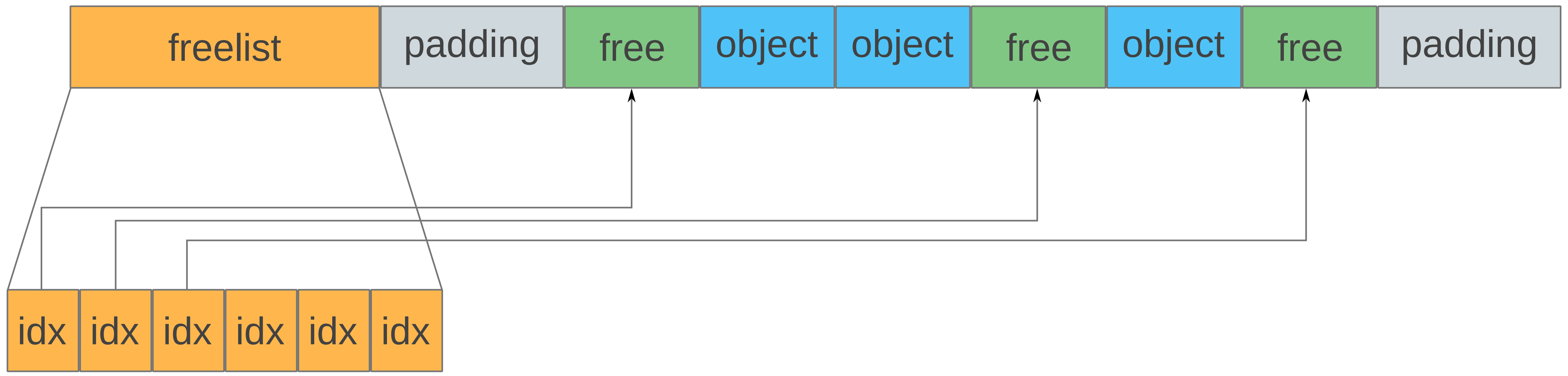
Added in Linux in 1996, implements work from Sun Microsystems in SunOS 5.4:
Jeff Bonwick, *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, **USENIX Summer 1994 Technical Conference**

- Cache friendly:
  - Queues to track per-cpu and per-node data
  - Page coloring to enforce cache locations
- NUMA aware: Per-node slabs, allocation are done on the local node by default, but other policies can be used depending on the objects stored
- Complex data structures, a lot of lists are needed for management

Design: Page frame layout

Metadata of each slab can be embedded in the slab itself:

- *freelist* contains the indexes of the free objects in this slab. It has as many entries as the number of objects that can be stored in the page frame
- *padding* aligns the objects properly



Note

Multiple allocations can be done by only touching one cache line (the one with the freelist). No need to touch the actual objects.

SLAB Allocator (2)

Design: Data structures

Partial description, see their real definitions for more details

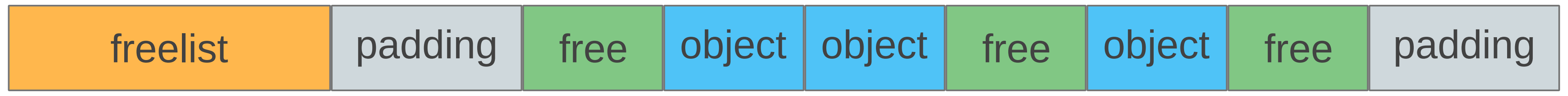
```
struct kmem_cache {
    struct array_cache __percpu *cpu_cache;
    unsigned int size;
    int object_size;
    struct kmem_cache_node *node[MAX_NUMNODES];
};
```

```
struct kmem_cache_node {
    struct list_head slabs_partial; // slabs with free objects
    struct list_head slabs_full;    // full slabs
    struct list_head slabs_free;    // empty slabs
    /* some counters, locks, and pointers to array_caches */
};
```

```
struct array_cache {
    unsigned int avail; // number of active entries
    unsigned int limit; // max number of entries
    void *entry[];      // LIFO array of free elements
};
```

```
struct slab {
    /* metadata */
    struct kmem_cache *slab_cache;
    struct list_head slab_list;
    void *freelist;
    void *s_mem;
    /* ... */
    /* actual data, containing the page frame layout shown here*/
};
```

Frame layout:



Object layout:

SLUB Allocator

Introduced in 2007. The idea is to simplify the implementation, with less queues.

Locality by having per-cpu slabs, still NUMA aware

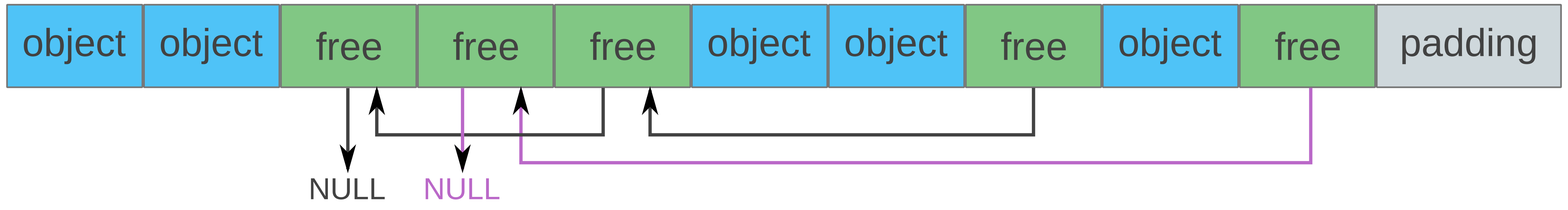
```
struct kmem_cache {
    struct kmem_cache_cpu __percpu *cpu_slab;
    unsigned int size;
    unsigned int object_size;
    struct kmem_cache_node *node[MAX_NUMNODES];
};
```

```
struct kmem_cache_node {
    unsigned long nr_partial;
    struct list_head partial;
};
```

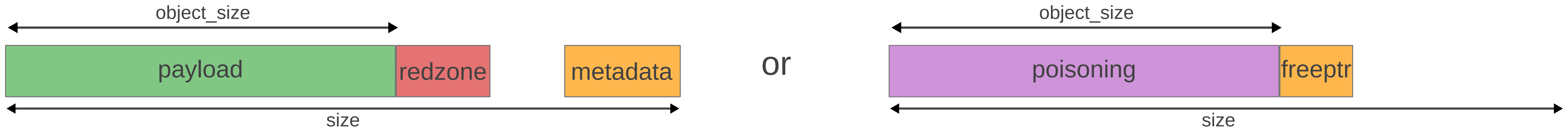
```
struct kmem_cache_cpu {
    void **freelist;
    struct slab *slab;
};
```

```
struct slab {
    struct kmem_cache *slab_cache;
    struct list_head slab_list;
    void *freelist;
    unsigned long counters; // nr_objects, nr_inuse
};
```

Frame layout:



Object layout:



Current State of Slab Allocators

SLUB is the default allocator

SLOB has been deprecated in 6.2

SLAB has been deprecated in 6.5

Manipulating Slabs

As seen previously, the “main” object of the slab layer is a `struct kmem_cache`, as it represents an instance of a cache.

Creation

```
#define KMEM_CACHE (struct, flags)
// struct is the type of object that wil be stored in the cache
// flags are SLAB_POISON, SLAB_RED_ZONE and SLAB_HWCACHE_ALIGN
```

```
static struct kmem_cache *cache;

int fn(void)
{
    /* ... */
    cache = KMEM_CACHE(bio_crypt_ctx, 0);
    /* ... */
}
```

Destruction

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

Allocation/Free

Use these methods as a replacement for `kmalloc()` and `kfree()`.

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

Slab Information

You can query which caches have been created in your system from user space:

```
cat /proc/slabinfo
```

```
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <sharedfactor> : sla
ext4_groupinfo_4k   7656   7656    184   44    2 : tunables    0    0    0 : slabdata    174    174      0
ext4_fc_dentry_update      0      0     96   42    1 : tunables    0    0    0 : slabdata      0      0      0
ext4_inode_cache  257751 258039   1192   27    8 : tunables    0    0    0 : slabdata   9557   9557      0
ext4_allocation_context    520    520    152   26    1 : tunables    0    0    0 : slabdata     20     20      0
ext4_prealloc_space    720    720    112   36    1 : tunables    0    0    0 : slabdata     20     20      0
ext4_io_end         1408   1408     64   64    1 : tunables    0    0    0 : slabdata     22     22      0
filp               17434  19008    256   32    2 : tunables    0    0    0 : slabdata    594    594      0
inode_cache        15325  15325    648   25    4 : tunables    0    0    0 : slabdata    613    613      0
dentry            357252 357252    192   42    2 : tunables    0    0    0 : slabdata   8506   8506      0
pid                 4129   4160    128   32    1 : tunables    0    0    0 : slabdata    130    130      0
kmalloc-8k           496    496   8192    4    8 : tunables    0    0    0 : slabdata    124    124      0
kmalloc-4k          1765   1768   4096    8    8 : tunables    0    0    0 : slabdata    221    221      0
kmalloc-2k          2452   2496   2048   16    8 : tunables    0    0    0 : slabdata    156    156      0
kmalloc-1k          4670   4704   1024   32    8 : tunables    0    0    0 : slabdata    147    147      0
kmalloc-512        49888  49888    512   32    4 : tunables    0    0    0 : slabdata   1559   1559      0
kmalloc-256        20977  21024    256   32    2 : tunables    0    0    0 : slabdata    657    657      0
kmalloc-192        53424  53424    192   42    2 : tunables    0    0    0 : slabdata   1272   1272      0
kmalloc-128        64768  64768    128   32    1 : tunables    0    0    0 : slabdata   2024   2024      0
kmalloc-96          7856   8820     96   42    1 : tunables    0    0    0 : slabdata    210    210      0
kmalloc-64         69111  69120     64   64    1 : tunables    0    0    0 : slabdata   1080   1080      0
kmalloc-32         26610  27008     32  128    1 : tunables    0    0    0 : slabdata    211    211      0
kmalloc-16         40386  41472     16  256    1 : tunables    0    0    0 : slabdata    162    162      0
kmalloc-8          32767  32768      8  512    1 : tunables    0    0    0 : slabdata     64     64      0
kmem_cache_node      640    640     64   64    1 : tunables    0    0    0 : slabdata     10     10      0
kmem_cache           384    384    256   32    2 : tunables    0    0    0 : slabdata     12     12      0
```

Memory Pools

If your code performs allocations and **needs** a guarantee that memory will be available, you can use **memory pools**.
This should be used only if your code will fail if memory is not available.
For example, some drivers performing DMA might need to allocate objects during an operation with hardware, where failure would break the hardware.

A *memory pool* is a chunk of pre-allocated memory that is guaranteed to be able to store at least a minimal number of objects.

Creation/Destruction

```
/**
 * mempool_create - create a memory pool
 * @min_nr:    the minimum number of elements guaranteed to be
 *             allocated for this pool.
 * @alloc_fn:  user-defined element-allocation function.
 * @free_fn:   user-defined element-freeing function.
 * @pool_data: optional private data available to the user-defined functions.
 */
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t *free_fn, void *pool_data);
void mempool_destroy(mempool_t *pool)
```

```
typedef void * (mempool_alloc_t)(gfp_t gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

Allocation/Free

```
void *mempool_alloc(mempool_t *pool, gfp_t gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

Memory pool on top of a slab cache

You can also build a memory pool on top of a slab cache with the following wrapper function:

```
static inline mempool_t *mempool_create_slab_pool(int min_nr, struct kmem_cache *kc);
```

If you want to use the `kmalloc` slab cache, you can use these wrapper function:

```
static inline mempool_t *mempool_create_kmalloc_pool(int min_nr, size_t size);
```

Freeing Memory

If you allocate memory, you **need** to free it at some point to avoid memory leaks.

You have three main ways of reclaiming the memory for the kernel:

- **Manually** free the memory you allocated when you don't need it anymore, *e.g.,* with `kfree()`.
  This works well for data that is allocated and freed in a single code path and easy to manage.
- Using **reference counters** to free objects when there are no more references to it.
  Works well when objects can be used in different subsystems or by different actors in the kernel.
- Memory reclamation by the **shrinker** under memory pressure.
  Should be set up for objects that may take a large portion of memory but are not necessary for your code to work, *e.g.,* you store statistics in memory without limiting the total amount of data you keep, but you are ok with freeing the old data if the system needs memory.

Reference Counters

Reference counters keep track of the number of users of an object.

Whenever the counter reaches 0, the object is not in use anymore and can be freed.

To use a reference counter, you need to embed a `struct kref` into your structure.
It **needs to be embedded**, not a pointer to one.

```
struct kref {
    refcount_t refcount; // this is a struct refcount_struct that contains an atomic_t which in turn is an int
};
```

You can check the full API in `include/linux/kref.h`, but the main methods are:

```
/**
 * kref_get - increment refcount for object.
 * @kref: object.
 */
void kref_get(struct kref *kref);

/**
 * kref_put - decrement refcount for object.
 * @kref: object.
 * @release: pointer to the function that will clean up the object when the
 *        last reference to the object is released.
 *        This pointer is required, and it is not acceptable to pass kfree
 *        in as this function.  If the caller does pass kfree to this
 *        function, you will be publicly mocked mercilessly by the kref
 *        maintainer, and anyone else who happens to notice it.  You have
 *        been warned.
 */
int kref_put(struct kref *kref, void (*release)(struct kref *kref));
```

The `void release(struct kref *kref)` function needs to free the object containing the *kref*.

You can achieve this by using the `container_of` macro.

Shrinker

If you allocate a lot of objects that are useful but **not necessary**, you can play nice and let the kernel reclaim your memory if needed.
When the kernel is under memory pressure, it runs the **shrinker** to reclaim memory from registered components.

To register a *shrinker* for your code, you first need to declare a `struct shrinker` and define a `count()` and a `scan()` functions.

```
struct shrinker {
    unsigned long (*count_objects)(struct shrinker *, struct shrink_control *sc);
    unsigned long (*scan_objects)(struct shrinker *, struct shrink_control *sc);

    int seeks;  /* seeks to recreate an obj */
    long batch; /* reclaim batch size, 0 = default */
    unsigned long flags;

    /* These are for internal use */
    struct list_head list;
    /* objs pending delete, per node */
    atomic_long_t *nr_deferred;
};
```

When the kernel wants to reclaim memory, it will call the `count()` method of all registered shrinkers to assess how many objects can be freed.
It will then call the `scan()` method if the count is positive in order to actually free the memory.

With your shrinker declared, you still need to register it so that the kernel will call it when memory reclaiming is performed.
You also need to unregister your shrinker when it is not usable anymore, *e.g.*, when you unload your module.

```
int register_shrinker(struct shrinker *);
void unregister_shrinker(struct shrinker *);
```

Resources

- Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB, Christoph Lameter, LinuxCon 2014
- **Linux Kernel Development, Third Edition**, Robert Love (Publisher: Addison-Wesley Professional), 2010