Atomik Kernel API

SLAB allocator

The primary memory allocation mechanism is the buddy allocator. However, the smallest allocatable memory unit of this allocator is exactly one page long (which, in x86 systems, is 4 KiB).

To reduce memory wasting by directly calling page_alloc and reduce object initialization overhead, Atomik provides the SLAB allocator abstraction, which optimizes the construction and destruction of objects of the same type. This can be understood as an application of object pool pattern to kernel space.

We can define a SLAB as an object with enough information to build an object of a defined type. A SLAB consists of one or more pages holding a SLAB header and an array of slots where preallocated objects are placed. The SLAB header contains a set of fields describing its occupancy, and pointers to constructor and destructor routines (if any).

All SLABs in the system are referenced by a global array and identified by a unique, up to 63 characters long ASCII name.

Structures

```
struct kmem_cache;
```

This is the basic structure used by the SLAB allocator, and describes a single SLAB. All allocations and freeings are performed against this object.

Functions

`struct kmem_cache *kmem_cache_lookup (const char *name);`

Searches a SLAB with a specified `name` in the system-wide list of SLABs. This function returns a pointer to the corresponding SLAB or `NULL` if the specified SLAB doesn't exist.

This function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes

`struct kmem_cache *kmem_cache_create (const char *name, busword_t size, void (*constructor) (struct kmem_cache *, void *), void (*destructor) (struct kmem_cache *, void *));`

Creates a SLAB named `name`, for allocating objects of size up to `size`, with an optional `constructor` and `destructor` for each object.

The function returns a pointer to the created SLAB, or `NULL` if there wasn't enough memory left to perform the allocation.

Upon the preallocation of any object, the function specified by `constructor` is called. If the SLAB needs to be shrunk, destructor is called for each object that is about to be freed. Both functions receive two arguments : a pointer to the refering SLAB and an opaque `void *` pointer with private data to be used by them. If the constructor is `NULL`, initialized objects are filled with zeros. If the destructor is `NULL`, no destruction operation is performed before freeing.

This function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : no
Thread-safe : yes
Interrupt-safe : yes

`void kmem_cache_set_opaque (struct kmem_cache *cache, void *private);`

Sets the opaque pointer passed to constructor and destructor of the SLAB `cache` to the value `private`.

This function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes

`void *__kmem_cache_alloc (struct kmem_cache *cache);`

Returns a pointer to a preallocated object in the SLAB `cache` and marks its slot as used. If there are no preallocated free slots, `__kmem_cache_alloc` returns `NULL`.

Note that this function lacks of any locking mechanism. It's the lowest-level SLAB allocation function and should only be used if the exclusive access to the SLAB is ensured.

Reentrant : yes
Thread-safe : no
Interrupt-safe : no

`void *kmem_cache_alloc_irq (struct kmem_cache *cache);`

Same as `__kmem_cache_alloc`, but ensures that no interrupts may be served during its execution. Thus, this function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes

`void *kmem_cache_alloc_task (struct kmem_cache *cache);`

Same as `__kmem_cache_alloc`, but protects the SLAB with its own private kernel mutex. Tasks that try to access a locked SLAB will sleep until it's unlocked by the owning task. As this function can sleep, it mustn't be called from interrupt context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : no

`void *kmem_cache_alloc (struct kmem_cache *cache);`

Same as `__kmem_cache_alloc` but detects the current context (task or interrupt) and calls `kmem_cache_alloc_task` or `kmem_cache_alloc_irq` accordingly.

These functions are inherently dangerous when misused. Calls to `kmem_cache_*_task` and `kmem_cache_*_irq` shouldn't never be merged against the same SLAB as interrupts will bypass the mutex lock and using `kmem_cache_*` makes more difficult to spot this kind of bugs. If allocations are going to happen both in interrupt and task context, in case of doubt, the best approach is to use `kmem_cache_*_irq` in both contexts.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes

`void __kmem_cache_free (struct kmem_cache *cache, void *ptr);`

Marks the slot corresponding to the object pointed by `ptr` as free in the SLAB `cache`.

Note that this function lacks of any locking mechanism. It's the lowest-level SLAB freeing function and should only be used if the exclusive access to the SLAB is ensured.

Reentrant : yes
Thread-safe : no
Interrupt-safe : no

`void kmem_cache_free_irq (struct kmem_cache *cache, void *ptr);`

Same as `__kmem_cache_free`, but ensures that no interrupts may be served during its execution. Thus, this function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes

`void kmem_cache_free_task (struct kmem_cache *cache, void *ptr);`

Same as `__kmem_cache_free`, but protects the SLAB with its own private kernel mutex. Tasks that try to access a locked SLAB will sleep until it's unlocked by the owning task. As this function can sleep, it mustn't be called from interrupt context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : no

`void kmem_cache_free (struct kmem_cache *cache, void *ptr);`

Same as `__kmem_cache_free` but detects the current context (task or interrupt) and calls `kmem_cache_free_task` or `kmem_cache_free_irq` accordingly.

These functions are inherently dangerous when misused. Calls to `kmem_cache_*_task` and `kmem_cache_*_irq` shouldn't never be merged against the same SLAB as interrupts will bypass the mutex lock and using `kmem_cache_*` makes more difficult to spot this kind of bugs. If allocations are going to happen both in interrupt and task context, in case of doubt, the best approach is to use `kmem_cache_*_irq` in both contexts.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes

`int __kmem_cache_grow (struct kmem_cache *cache);`

Adds more free pages to SLAB `cache`. This function is usually called immediately after a failure of `kmem_cache_alloc` to get more storage space available. Each call to this function will try to grow twice as big as the previous successful growth operation against the SLAB. This function returns `0` on success or `-1` if there weren't enough free pages in the system to make the SLAB grow.

Note that this function lacks of any locking mechanism. It's the lowest-level SLAB growth function and should only be used if the exclusive access to the SLAB is ensured.

Reentrant : yes
Thread-safe : no
Interrupt-safe : no

`int kmem_cache_grow_irq (struct kmem_cache *cache)`

Same as `__kmem_cache_grow`, but ensures that no interrupts may be served during its execution. Thus, this function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes

`int kmem_cache_grow_task (struct kmem_cache *cache)`

Same as `__kmem_cache_grow`, but protects the SLAB with its own private kernel mutex. Tasks that try to access a locked SLAB will sleep until it's unlocked by the owning task. As this function can sleep, it mustn't be called from interrupt context.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : no

`int kmem_cache_grow (struct kmem_cache *cache, void *ptr)`

Same as `__kmem_cache_grow` but detects the current context (task or interrupt) and calls `kmem_cache_grow_task` or `kmem_cache_grow_irq` accordingly.

These functions are inherently dangerous when misused. Calls to `kmem_cache_*_task` and `kmem_cache_*_irq` shouldn't never be merged against the same SLAB as interrupts will bypass the mutex lock and using `kmem_cache_*` makes more difficult to spot this kind of bugs. If allocations are going to happen both in interrupt and task context, in case of doubt, the best approach is to use `kmem_cache_*_irq` in both contexts.

Reentrant : yes
Thread-safe : yes
Interrupt-safe : yes