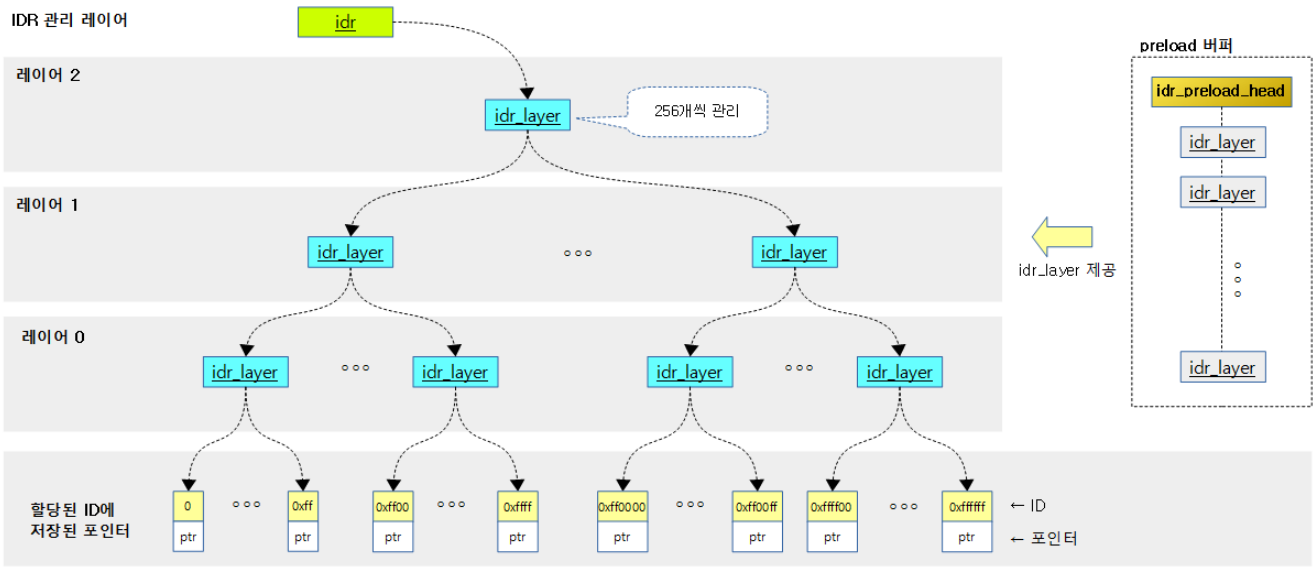
Integer Identity Management (IDR)

2016-08-26 (http://jake.dothome.co.kr/idr/) Moon Young-il (http://jake.dothome.co.kr/author/admin/)
Linux Kernel (http://jake.dothome.co.kr/category/linux/)

Use a radix tree to manage integer IDs and return pointer values associated with them. The following are the features of Linux IDR.

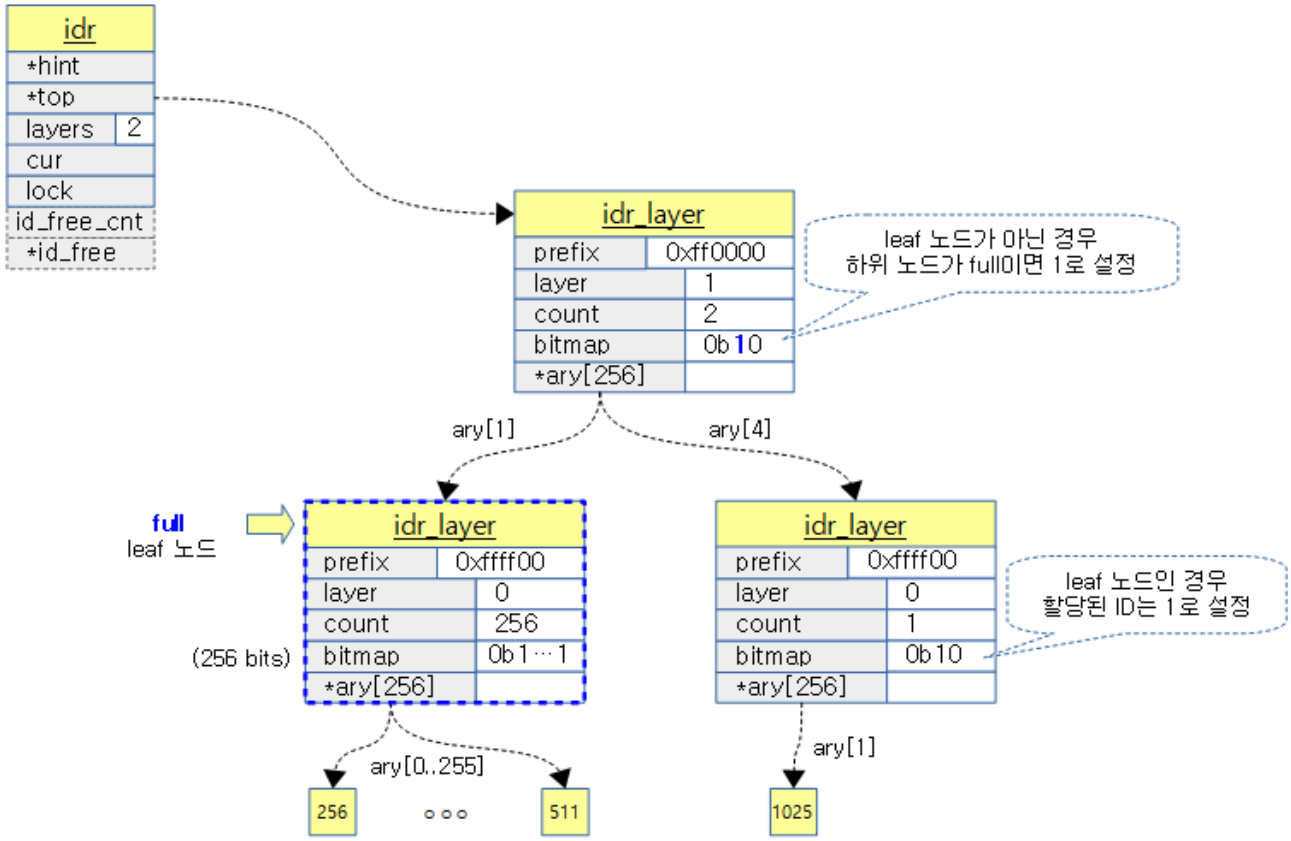
- Identity Management
  - You can use a radix tree to manage IDs in 256 (0x100) increments for each layer step.
    - Depending on the number of layers used by the 32-bit system
      - 1 layer: 0 ~ 0xff identity management
      - 2 layers: 0 ~ 0xffff identity management
      - 3 layers: 0 ~ 0xffffff identity management
      - 4 layers: 0 ~ 0x7fffffff identity management
    - Depending on the number of layers used by the 64-bit system
      - 1 layer: 0 ~ 0xff identity management
      - 2 layers: 0 ~ 0xffff identity management
      - 3 layers: 0 ~ 0xffffff identity management
      - 4 layers: 0 ~ 0xffffffff identity management
      - 5 layers: 0 ~ 0xff_ffffffff identity management
      - 6 layers: 0 ~ 0xffff_ffffffff identity management
      - 7 layers: 0 ~ 0xffffff_ffffffff identity management
      - 8 layers: 0 ~ 0x7fffffff_ffffffff identity management
  - If you require a large number of IDs, the layer management step will be larger, which will cost you more.
- IDR preload buffer
  - idr_layer struct allocation is passed through the slub cache, which is designed to be allocated several in advance so that it can be quickly provisioned when the IDR layer expands horizontally or vertically.

The following illustration shows how IDR is managed by layers.



(http://jake.dothome.co.kr/wp-content/uploads/2016/08/idr-2.png)

The following figure shows how IDs 256~511 and IDs 1025 are assigned and managed.



(http://jake.dothome.co.kr/wp-content/uploads/2016/08/idr-1.png)

Declaring and initializing static IDR

DEFINE_IDR()

include/linux/idr.h

```
1 #define DEFINE_IDR(name)        struct idr name = IDR_INIT(name)
```

Declare and initialize the idr struct with the given name.

IDR_INIT()

include/linux/idr.h

```
#define IDR_INIT(name)
\
{
\
        .lock                   = __SPIN_LOCK_UNLOCKED(name.lock),
\
}
```

Initialize the idr struct with the given name.

Initialize dynamic IDR

idr_init()

```
/**
 * idr_init - initialize idr handle
 * @idp:        idr handle
 *
 * This function is use to set up the handle (@idp) that you will pass
 * to the rest of the functions.
 */
void idr_init(struct idr *idp)
{
        memset(idp, 0, sizeof(struct idr));
        spin_lock_init(&idp->lock);
}
EXPORT_SYMBOL(idr_init);
```

Initialize all idr struct member variables to 0 and only the lock member to spinlock.

IDR Allocation

- In the case of assigning IDs in the old way, the following two APIs were used in succession:
  - int idr_pre_get(struct idr *idp, gfp_t gfp_mask);
  - int idr_get_new(struct idr *idp, void *ptr, int *id);
- If you're assigning IDs using the new method, use the following three APIs in succession:
  - void idr_preload(gfp_t gfp_mask);
  - int idr_alloc(struct idr *idp, void *ptr, int start, int end, gfp_t gfp_mask);
  - void idr_preload_end(void);
  - Applies as of kernel v2013.3-rc9 in 1
    - 참고: idr: implement idr_preload[_end]() and idr_alloc() (https://github.com/torvalds/linux/commit/d5c7409f79e14db49d0078569233465759 2c07ff)

idr_preload()

lib/idr.c

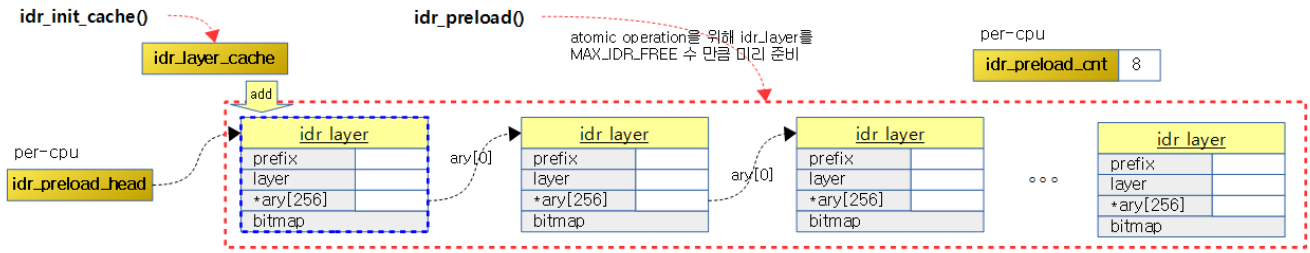
```
/**
 * idr_preload - preload for idr_alloc()
 * @gfp_mask: allocation mask to use for preloading
 *
 * Preload per-cpu layer buffer for idr_alloc().  Can only be used from
 * process context and each idr_preload() invocation should be matched with
 * idr_preload_end().  Note that preemption is disabled while preloaded.
 *
 * The first idr_alloc() in the preloaded section can be treated as if it
 * were invoked with @gfp_mask used for preloading.  This allows using more
 * permissive allocation masks for idrs protected by spinlocks.
 *
 * For example, if idr_alloc() below fails, the failure can be treated as
 * if idr_alloc() were called with GFP_KERNEL rather than GFP_NOWAIT.
 *
 *      idr_preload(GFP_KERNEL);
 *      spin_lock(lock);
 *
 *      id = idr_alloc(idr, ptr, start, end, GFP_NOWAIT);
 *
 *      spin_unlock(lock);
 *      idr_preload_end();
 *      if (id < 0)
 *              error;
 */
void idr_preload(gfp_t gfp_mask)
{
        /*
         * Consuming preload buffer from non-process context breaks preload
         * allocation guarantee.  Disallow usage from those contexts.
         */
        WARN_ON_ONCE(in_interrupt());
        might_sleep_if(gfp_mask & __GFP_WAIT);

        preempt_disable();

        /*
         * idr_alloc() is likely to succeed w/o full idr_layer buffer and
         * return value from idr_alloc() needs to be checked for failure
         * anyway.  Silently give up if allocation fails.  The caller can
         * treat failures from idr_alloc() as if idr_alloc() were called
         * with @gfp_mask which should be enough.
         */
        while (__this_cpu_read(idr_preload_cnt) < MAX_IDR_FREE) {
                struct idr_layer *new;

                preempt_enable();
                new = kmem_cache_zalloc(idr_layer_cache, gfp_mask);
                preempt_disable();
                if (!new)
                        break;

                /* link the new one to per-cpu preload list */
                new->ary[0] = __this_cpu_read(idr_preload_head);
                __this_cpu_write(idr_preload_head, new);
                __this_cpu_inc(idr_preload_cnt);
        }
```

```
58 }
59 EXPORT_SYMBOL(idr_preload);
```

Pre-allocate 8 (on 32-bit systems) idr_layer entries in the idr preload buffer. This IDR preload buffer is used to prepare the idr_layer required by the idr_alloc() function in advance and provide it whenever it is needed in the shortest possible time, so that the preemption disable period is shortened as much as possible for the idr_alloc() function to run in the preemption disable section.

- might_sleep_if(gfp_mask & __GFP_WAIT);
  - Given a __GFP_WAIT flag, it can be preempted if there is a task to be processed with a higher priority than the current task. In other words, sleep is possible.
- preempt_disable();
  - Don't get preempted from here.
- while (__this_cpu_read(idr_preload_cnt) < MAX_IDR_FREE) {
  - If the current CPU has less than MAX_IDR_FREE idr_preload_cnt (twice the maximum level), it will continue to loop.
    - The purpose is to allocate an IDR cache in advance for MAX_IDR_FREE (twice the maximum level).
- preempt_enable(); new = kmem_cache_zalloc(idr_layer_cache, gfp_mask); preempt_disable(); if (!new) break;
  - With the preemption enabled, the idr_layer_cache is allocated a idr_layer struct area, and if it fails, it exits the loop.
- new->ary[0] = __this_cpu_read(idr_preload_head); __this_cpu_write(idr_preload_head, new);
  - Add the assigned new idr_layer struct to the idr preload buffer list.
    - idr_preload_head idr_layer in the list uses ary[0] to connect the next entry.
- __this_cpu_inc(idr_preload_cnt);
  - It increases the idr_preload_cnt by adding it.

The following figure shows that the idr_preload() function has been allocated 8 (32-bit systems) idr_layer structs in advance to the idr preload buffer.



(http://jake.dothome.co.kr/wp-content/uploads/2016/08/idr_preload-1.png)

idr_preload_end()

include/linux/idr.h

```
01 /**
02  * idr_preload_end - end preload section started with idr_preload()
03  *
04  * Each idr_preload() should be matched with an invocation of this
05  * function.  See idr_preload() for details.
```

```
 */
static inline void idr_preload_end(void)
{
        preempt_enable();
}
```

Now that idr_alloc() is done, we also enable the preemption to change the preemptible state as well.

idr_alloc()

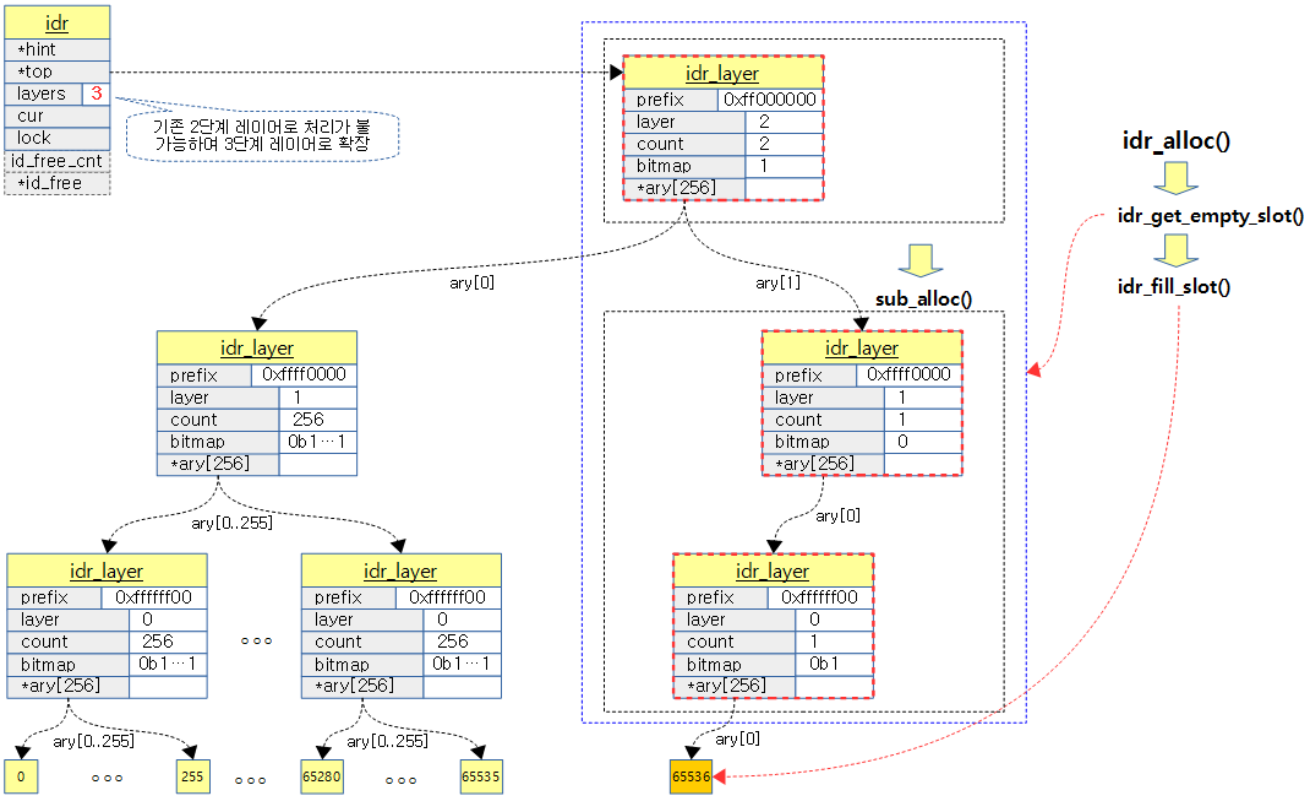
lib/idr.c

```
/**
 * idr_alloc - allocate new idr entry
 * @idr: the (initialized) idr
 * @ptr: pointer to be associated with the new id
 * @start: the minimum id (inclusive)
 * @end: the maximum id (exclusive, <= 0 for max)
 * @gfp_mask: memory allocation flags
 *
 * Allocate an id in [start, end) and associate it with @ptr.  If no ID is
 * available in the specified range, returns -ENOSPC.  On memory allocation
 * failure, returns -ENOMEM.
 *
 * Note that @end is treated as max when <= 0.  This is to always allow
 * using @start + N as @end as long as N is inside integer range.
 *
 * The user is responsible for exclusively synchronizing all operations
 * which may modify @idr.  However, read-only accesses such as idr_find()
 * or iteration can be performed under RCU read lock provided the user
 * destroys @ptr in RCU-safe way after removal from idr.
 */
int idr_alloc(struct idr *idr, void *ptr, int start, int end, gfp_t gfp_mask)
{
        int max = end > 0 ? end - 1 : INT_MAX;  /* inclusive upper limit */
        struct idr_layer *pa[MAX_IDR_LEVEL + 1];
        int id;

        might_sleep_if(gfpflags_allow_blocking(gfp_mask));

        /* sanity checks */
        if (WARN_ON_ONCE(start < 0))
                return -EINVAL;
        if (unlikely(max < start))
                return -ENOSPC;

        /* allocate id */
        id = idr_get_empty_slot(idr, start, pa, gfp_mask, NULL);
        if (unlikely(id < 0))
                return id;
        if (unlikely(id > max))
                return -ENOSPC;

        idr_fill_slot(idr, ptr, id, pa);
        return id;
}
EXPORT_SYMBOL_GPL(idr_alloc);
```

Find an empty id within the range of start ~ (end-1) integers, store the ptr, and return the id. If END is 0, it is specified as INT_MAX, which is the maximum integer value of the system.

- might_sleep_if(gfp_mask & __GFP_WAIT);
  - If a __GFP_WAIT flag is requested while preemptible, sleep if the highest-ranked task requests preemption.
- id = idr_get_empty_slot(idr, start, pa, gfp_mask, NULL);
  - Find and return the empty ID from start ~ end-1. In the process, you will be able to create a layer if you need to expand it.
- if (unlikely(id < 0)) return id;
  - If it returns a negative number, it returns an error because no ID is assigned.
- if (unlikely(id > max)) return -ENOSPC;
  - If allocation is not possible within the scope of the request, return the -ENOSPC error that there is no space to allocate.
- idr_fill_slot(idr, ptr, id, pa);
  - Update each layer that corresponds to the id.

The following figure shows how the IDR of 0 is additionally allocated and expanded to the 65535rd level layer while the ID of 2~65536 is full and managed as a 3nd level layer.



(http://jake.dothome.co.kr/wp-content/uploads/2016/08/idr_alloc-1.png)

idr_get_empty_slot()

lib/idr.c

```
01 static int idr_get_empty_slot(struct idr *idp, int starting_id,
02                                struct idr_layer **pa, gfp_t gfp_mask,
```

```
                                struct idr *layer_idr)
{
        struct idr_layer *p, *new;
        int layers, v, id;
        unsigned long flags;

        id = starting_id;
build_up:
        p = idp->top;
        layers = idp->layers;
        if (unlikely(!p)) {
                if (!(p = idr_layer_alloc(gfp_mask, layer_idr)))
                        return -ENOMEM;
                p->layer = 0;
                layers = 1;
        }
        /*
         * Add a new layer to the top of the tree if the requested
         * id is larger than the currently allocated space.
         */
        while (id > idr_max(layers)) {
                layers++;
                if (!p->count) {
                        /* special case: if the tree is currently empty,
                         * then we grow the tree by moving the top node
                         * upwards.
                         */
                        p->layer++;
                        WARN_ON_ONCE(p->prefix);
                        continue;
                }
                if (!(new = idr_layer_alloc(gfp_mask, layer_idr))) {
                        /*
                         * The allocation failed.  If we built part of
                         * the structure tear it down.
                         */
                        spin_lock_irqsave(&idp->lock, flags);
                        for (new = p; p && p != idp->top; new = p) {
                                p = p->ary[0];
                                new->ary[0] = NULL;
                                new->count = 0;
                                bitmap_clear(new->bitmap, 0, IDR_SIZE);
                                __move_to_free_list(idp, new);
                        }
                        spin_unlock_irqrestore(&idp->lock, flags);
                        return -ENOMEM;
                }
                new->ary[0] = p;
                new->count = 1;
                new->layer = layers-1;
                new->prefix = id & idr_layer_prefix_mask(new->layer);
                if (bitmap_full(p->bitmap, IDR_SIZE))
                        __set_bit(0, new->bitmap);
                p = new;
        }
        rcu_assign_pointer(idp->top, p);
        idp->layers = layers;
        v = sub_alloc(idp, &id, pa, gfp_mask, layer_idr);
        if (v == -EAGAIN)
                goto build_up;
        return(v);
}
```

Find and return the empty ID from start ~ end-1. In this process, if you need to extend the layer (tree depth), you can create it.

- build_up: p = idp->top; layers = idp->layers;

    - Specify the node that the top of the idr structure points to, and know how many layer layers it uses.
- if (unlikely(!p)) { if (!(p = idr_layer_alloc(gfp_mask, layer_idr))) return -ENOMEM;
    - If there is a small probability that no node is specified and no idr_layer structure is assigned, it returns a -ENOMEM error.
- p->layer = 0; layers = 1;
    - Since it's a leaf node, we assign 0 to the layer member variable, and 1 to the number of layers.
- while (id > idr_max(layers)) {
    - If the requested id values exceed the maximum number that the current IDR layer can handle, it goes into a loop.
- if (!(new = idr_layer_alloc(gfp_mask, layer_idr))) {
    - Expand the upper layer (tree depth) and the allocation fails.
- for (new = p; p && p != idp->top; new = p) { p = p->ary[0]; new->ary[0] = NULL; new->count = 0; bitmap_clear(new->bitmap, 0, IDR_SIZE); __move_to_free_list(idp, new); }
    - Move all the idr_layer you've already created to expand the layer to the id_free list.
- new->ary[0] = p; new->count = 1; new->layer = layers-1; new->prefix = id & idr_layer_prefix_mask(new->layer); if (bitmap_full(p->bitmap, IDR_SIZE)) __set_bit(0, new->bitmap); p = new;
    - Assign ary[0] of the newly created layer so that it faces the existing layer, and assign it so that it has 1 count.
    - Specify a prefix value, and if the existing layer's bitmap is full, set the first bit of the newly created bitmap to full to 1.
- rcu_assign_pointer(idp->top, p);
    - Assign so that the top of the idr structure can point to the p node.
- idp->layers = layers;
    - Update the layers of the idr structure.
- v = sub_alloc(idp, &id, pa, gfp_mask, layer_idr);
    - Extending the upper layer tree depth is completed in the while statement above, where we assign an ID, but without changing the depth of the layer, we assign the layers below that range that need to be assigned.
- if (v == -EAGAIN) goto build_up;
    - If identity assignment fails, start over from build_up.

idr_layer_alloc()

lib/idr.c

```
01 /**
02  * idr_layer_alloc - allocate a new idr_layer
03  * @gfp_mask: allocation mask
04  * @layer_idr: optional idr to allocate from
05  *
06  * If @layer_idr is %NULL, directly allocate one using @gfp_mask or fetc
    h
07  * one from the per-cpu preload buffer.  If @layer_idr is not %NULL, fet
    ch
08  * an idr_layer from @idr->id_free.
```

```
 *
 * @layer_idr is to maintain backward compatibility with the old alloc
 * interface - idr_pre_get() and idr_get_new*() - and will be removed
 * together with per-pool preload buffer.
 */
static struct idr_layer *idr_layer_alloc(gfp_t gfp_mask, struct idr *lay
er_idr)
{
        struct idr_layer *new;

        /* this is the old path, bypass to get_from_free_list() */
        if (layer_idr)
                return get_from_free_list(layer_idr);

        /*
         * Try to allocate directly from kmem_cache.  We want to try thi
s
         * before preload buffer; otherwise, non-preloading idr_alloc()
         * users will end up taking advantage of preloading ones.  As th
e
         * following is allowed to fail for preloaded cases, suppress
         * warning this time.
         */
        new = kmem_cache_zalloc(idr_layer_cache, gfp_mask | __GFP_NOWAR
N);
        if (new)
                return new;

        /*
         * Try to fetch one from the per-cpu preload buffer if in proces
s
         * context.  See idr_preload() for details.
         */
        if (!in_interrupt()) {
                preempt_disable();
                new = __this_cpu_read(idr_preload_head);
                if (new) {
                        __this_cpu_write(idr_preload_head, new->ary[0]);
                        __this_cpu_dec(idr_preload_cnt);
                        new->ary[0] = NULL;
                }
                preempt_enable();
                if (new)
                        return new;
        }

        /*
         * Both failed.  Try kmem_cache again w/o adding __GFP_NOWARN so
         * that memory allocation failure warning is printed as intende
d.
         */
        return kmem_cache_zalloc(idr_layer_cache, gfp_mask);
}
```

It is allocated a idr_layer struct from the idr preload buffer or idr_layer_cache.

- if (layer_idr) return get_from_free_list(layer_idr);
  - If the second argument of this function, layer_idr, is not null, it fetches the idr_layer from the id_free member of the IDR structure to make it compatible with the existing allocation scheme.
  - The new way to use the idr preload buffer is to skip this routine because the layer_idr value is nulled.
- new = kmem_cache_zalloc(idr_layer_cache, gfp_mask | __GFP_NOWARN); if (new) return new;

    - In order to accommodate the use of the idr_alloc() function without the idr_preload() function, the idr_layer struct is allocated directly from the idr_layer_cache rather than the idr preload buffer.
        - If the idr_preload() function is used, this function is advanced so that if an error occurs, a warning message will not be printed by the __GFP_NOWARN option.
- if (!in_interrupt()) { preempt_disable();
    - Unless it is called from an interrupt handler, it blocks the preemptive point to use the idr preload buffer.
- new = __this_cpu_read(idr_preload_head); if (new) { __this_cpu_write(idr_preload_head, new->ary[0]); __this_cpu_dec(idr_preload_cnt); new->ary[0] = NULL; }
    - idr_preload_head takes the idr_layer struct from the list and removes it from the list.
- preempt_enable(); if (new) return new;
    - Return it back to the preemptible state and return it if the allocation is successful.
- return kmem_cache_zalloc(idr_layer_cache, gfp_mask);
    - If all else fails, try again one last time directly from the idr_layer_cache. In this case, a warning message is displayed in case of failure.

get_from_free_list()

lib/idr.c

```
01 static struct idr_layer *get_from_free_list(struct idr *idp)
02 {
03         struct idr_layer *p;
04         unsigned long flags;
05
06         spin_lock_irqsave(&idp->lock, flags);
07         if ((p = idp->id_free)) {
08                 idp->id_free = p->ary[0];
09                 idp->id_free_cnt--;
10                 p->ary[0] = NULL;
11         }
12         spin_unlock_irqrestore(&idp->lock, flags);
13         return(p);
14 }
```

id_free If there are entries in the list, remove them from the list and return them.

idr_max()

lib/idr.c

```
1 /* the maximum ID which can be allocated given idr->layers */
2 static int idr_max(int layers)
3 {
4         int bits = min_t(int, layers * IDR_BITS, MAX_IDR_SHIFT);
5
6         return (1 << bits) - 1;
7 }
```

Find out the max ID (positive integer) that can be assigned to a given layer.

- e.g. 32-bit system

- Level 1 Layer: 0xff (8bit)
- Level 2 layer: 0xffff (16bit)
- Level 3 layer: 0xff_ffff (24bit)
- Level 4 layer: 0x7fff_ffff (Positive Integer, limited to 31bit)

__move_to_free_list()

lib/idr.c

```
/* only called when idp->lock is held */
static void __move_to_free_list(struct idr *idp, struct idr_layer *p)
{
        p->ary[0] = idp->id_free;
        idp->id_free = p;
        idp->id_free_cnt++;
}
```

Add the entry to the id_free list.

idr_layer_prefix_mask()

lib/idr.c

```
/*
 * Prefix mask for an idr_layer at @layer.  For layer 0, the prefix mask is
 * all bits except for the lower IDR_BITS.  For layer 1, 2 * IDR_BITS, and
 * so on.
 */
static int idr_layer_prefix_mask(int layer)
{
        return ~idr_max(layer + 1);
}
```

Returns the prefix value for the request layer.

- e.g. if you request layer 0, return 0xffffff00.

sub_alloc()

lib/idr.c

```
/**
 * sub_alloc - try to allocate an id without growing the tree depth
 * @idp: idr handle
 * @starting_id: id to start search at
 * @pa: idr_layer[MAX_IDR_LEVEL] used as backtrack buffer
 * @gfp_mask: allocation mask for idr_layer_alloc()
 * @layer_idr: optional idr passed to idr_layer_alloc()
 *
 * Allocate an id in range [@starting_id, INT_MAX] from @idp without
 * growing its depth.  Returns
 *
 *  the allocated id >= 0 if successful,
 *  -EAGAIN if the tree needs to grow for allocation to succeed,
 *  -ENOSPC if the id space is exhausted,
 *  -ENOMEM if more idr_layers need to be allocated.
```

```
 */
static int sub_alloc(struct idr *idp, int *starting_id, struct idr_layer **pa,
                     gfp_t gfp_mask, struct idr *layer_idr)
{
        int n, m, sh;
        struct idr_layer *p, *new;
        int l, id, oid;

        id = *starting_id;
 restart:
        p = idp->top;
        l = idp->layers;
        pa[l--] = NULL;
        while (1) {
                /*
                 * We run around this while until we reach the leaf node...
                 */
                n = (id >> (IDR_BITS*l)) & IDR_MASK;
                m = find_next_zero_bit(p->bitmap, IDR_SIZE, n);
                if (m == IDR_SIZE) {
                        /* no space available go back to previous layer. */
                        l++;
                        oid = id;
                        id = (id | ((1 << (IDR_BITS * l)) - 1)) + 1;

                        /* if already at the top layer, we need to grow */
                        if (id > idr_max(idp->layers)) {
                                *starting_id = id;
                                return -EAGAIN;
                        }
                        p = pa[l];
                        BUG_ON(!p);

                        /* If we need to go up one layer, continue the
                         * loop; otherwise, restart from the top.
                         */
                        sh = IDR_BITS * (l + 1);
                        if (oid >> sh == id >> sh)
                                continue;
                        else
                                goto restart;
                }
```

Assign an ID, but don't change the depth of the layer, and assign the layers that need to be assigned among the lower layers within that range. The output argument pa pointer array contains the pointer addresses of idr_layer, from the top layer to the layer corresponding to the id.

- pa[0] points to the bottom layer, and then the array increases to the top layer related to the id, and ends with null at the end.

- id = *starting_id;
  - Prepare from starting_id.
- restart: p = idp->top;
  - If you need to do it all over again, go to the restart: label here and prepare the top-level node connected to the top of the idr structure.
- l = idp->layers; pa[l–] = NULL;

    - l knows how many layers it uses and assigns null to the end of the last pa[] array to be processed.
- while (1) { n = (id >> (IDR_BITS*l)) & IDR_MASK; m = find_next_zero_bit(p->bitmap, IDR_SIZE, n);
    - In a loop, n assigns the id value to the bitmap index n as the IDR_MASK value of the quotient divided by the current layer value x 8, and the bit position set to 0 after the n value in the bitmap is found to m. If it can't be found, it returns a IDR_SIZE value.
    - e.g. if the entire layer is level 3 and you want to be assigned the last remaining ID in the current level.
        - id=0, bitmap=0x7fffffff_ffffffff_ffffffff_ffffffff_ffffffff_ffffffff_ffffffff_ffffffff, l=2
            - n=0, m=0xff
- if (m == IDR_SIZE) {
    - If there is no empty space after the specified number, it means that there is no identity space on the current node to process.
- l++; old = id; id = (id | ((1 << (IDR_BITS * l)) – 1)) + 1;
    - Increase L to go back to the upper layer, back up the current id, and specify the first id of the right sibling layer in the current layer to find the next empty spot.
- if (id > idr_max(idp->layers)) { *starting_id = id; return -EAGAIN; }
    - If there is no more space to assign IDs with the current layer-level structure, assign id values to the starting_id and return -EAGAIN to request that the levels of the layer be expanded.
- p = pa[l];
    - Specify the top layer.
- sh = IDR_BITS * (l + 1); if (oid >> sh == id >> sh) continue; else goto restart;
    - If the newly assigned id can be processed by the parent node, it will continue the loop, otherwise it will go to the restart label and start all over again.
- if (m != n) { sh = IDR_BITS*l; id = ((id >> sh) ^ n ^ m) << sh; }
    - 예) id=0, n=0, m=0xff, l=2
        - id=0xff0000
- if ((id >= MAX_IDR_BIT) || (id < 0)) return -ENOSPC;
    - If the id value falls outside the range of positive integers, the system cannot handle it and returns a -ENOSPC error.
- if (l == 0) break;
    - If you've processed the last leaf node layer, exit the loop.
- if (!p->ary[m]) { new = idr_layer_alloc(gfp_mask, layer_idr); if (!new) return -ENOMEM;
    - If there are no child nodes, it is created.
- new->layer = l-1; new->prefix = id & idr_layer_prefix_mask(new->layer); rcu_assign_pointer(p->ary[m], new); p->count++;
    - Specify the layer and prefix of the child node, connect it to ary[] on the current node, and increment the count.

```
01                 if (m != n) {
02                         sh = IDR_BITS*l;
03                         id = ((id >> sh) ^ n ^ m) << sh;
04                 }
05                 if ((id >= MAX_IDR_BIT) || (id < 0))
```

```
                        return -ENOSPC;
                if (l == 0)
                        break;
                /*
                 * Create the layer below if it is missing.
                 */
                if (!p->ary[m]) {
                        new = idr_layer_alloc(gfp_mask, layer_idr);
                        if (!new)
                                return -ENOMEM;
                        new->layer = l-1;
                        new->prefix = id & idr_layer_prefix_mask(new->la
yer);
                        rcu_assign_pointer(p->ary[m], new);
                        p->count++;
                }
                pa[l--] = p;
                p = p->ary[m];
        }

        pa[l] = p;
        return id;
}
```

- if (m != n) { sh = IDR_BITS*l; id = ((id >> sh) ^ n ^ m) << sh; }
  - If m and n are different, find the ID of the vacant digit.
- if ((id >= MAX_IDR_BIT) || (id < 0)) return -ENOSPC;
  - If the id value is out of range, it returns an error that there is no space to allocate.
- if (l == 0) break;
  - If you get to the bottom layer, exit the loop.
- if (!p->ary[m]) { new = idr_layer_alloc(gfp_mask, layer_idr); if (!new) return -ENOMEM;
  - If there is no sublayer node to manage the ID number to be assigned, the layer is assigned.
- new->layer = l-1; new->prefix = id & idr_layer_prefix_mask(new->layer); rcu_assign_pointer(p->ary[m], new); p->count++;
  - Update the assigned layer number, prefix, count, etc., and point to the layer assigned to ary[m].
- pa[l–] = p; p = p->ary[m]; }
  - Then specify the reduction to process the layer below and continue the loop.
- pa[l] = p; return id;
  - Updates the last pa[0] and returns the id.

idr_fill_slot()

lib/idr.c

```
/*
 * @id and @pa are from a successful allocation from idr_get_empty_slot
().
 * Install the user pointer @ptr and mark the slot full.
 */
static void idr_fill_slot(struct idr *idr, void *ptr, int id,
                          struct idr_layer **pa)
{
        /* update hint used for lookup, cleared from free_layer() */
        rcu_assign_pointer(idr->hint, pa[0]);

```

```
        rcu_assign_pointer(pa[0]->ary[id & IDR_MASK], (struct idr_layer
*)ptr);
        pa[0]->count++;
        idr_mark_full(pa, id);
}
```

Assign the address of the leaf node to which you assigned the id at the end to the hint member of the idr structure, store the ptr in the ary[] array, increment the count, and set the bitmap of the layers to 1.

- rcu_assign_pointer(idr->hint, pa[0]);
  - Store the address of the leaf node that you assigned the id to at the end in the hint member of the idr structure.
  - When searching by id in the idr_find() function, it is used to expedite the search if the layer that the hint points to covers the requested id.
- rcu_assign_pointer(pa[0]->ary[id & IDR_MASK], (struct idr_layer *)ptr);
  - Store the ptr value in the ary[] array, which corresponds to the id of the leaf node to which you assigned the id at the end.
- pa[0]->count++;
  - Increment the counter of the leaf node that assigned the id at the end.
- idr_mark_full(pa, id);
  - Set the bit corresponding to the id to 1 on the bitmap of the leaf node that assigned the ID at the end to indicate that the ID has been assigned, and then set the bit to 1 on the bitmap of the parent node of the node that is full among the top nodes from that node onwards.

idr_mark_full()

lib/idr.c

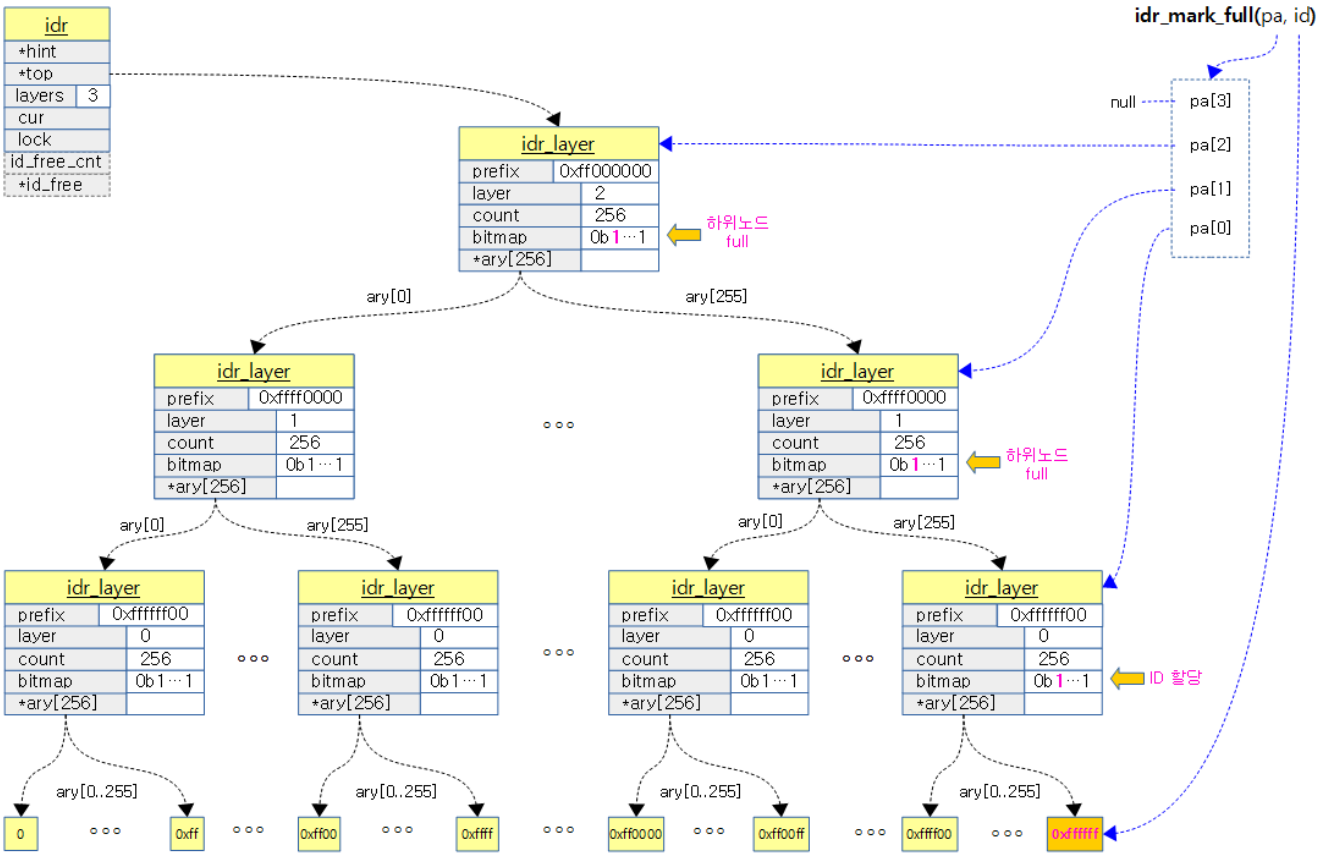
```
static void idr_mark_full(struct idr_layer **pa, int id)
{
        struct idr_layer *p = pa[0];
        int l = 0;

        __set_bit(id & IDR_MASK, p->bitmap);
        /*
         * If this layer is full mark the bit in the layer above to
         * show that this part of the radix tree is full.  This may
         * complete the layer above and require walking up the radix
         * tree.
         */
        while (bitmap_full(p->bitmap, IDR_SIZE)) {
                if (!(p = pa[++l]))
                        break;
                id = id >> IDR_BITS;
                __set_bit((id & IDR_MASK), p->bitmap);
        }
}
```

Set the bit corresponding to the id to 1 on the bitmap of the leaf node that assigned the ID at the end to indicate that the ID has been assigned, and then set the bit to 1 on the bitmap of the parent node of the node that is full among the top nodes from that node onwards.

- struct idr_layer *p = pa[0];
  - Lowest Leaf Node

- __set_bit(id & IDR_MASK, p->bitmap);
  - Set the position corresponding to the id in the bitmap of the idr_layer node to 1 to indicate that the ID has been assigned.
- while (bitmap_full(p->bitmap, IDR_SIZE)) {
  - If the node is full, it will continue to loop.
- if (!(p = pa[++l])) break;
  - If the parent node is not specified, it exits the loop.
- id = id >> IDR_BITS;
  - Divide the id value by 256.
- __set_bit((id & IDR_MASK), p->bitmap);
  - Set the position corresponding to the id in the current node's Bitmap to 1 to indicate that the child node has become full.

The following figure shows how each bitmap is fully processed by the idr_mark_full() function after being assigned a 0xffffff ID.



(http://jake.dothome.co.kr/wp-content/uploads/2016/08/idr_mark_full-1.png)

Release IDR

idr_remove()

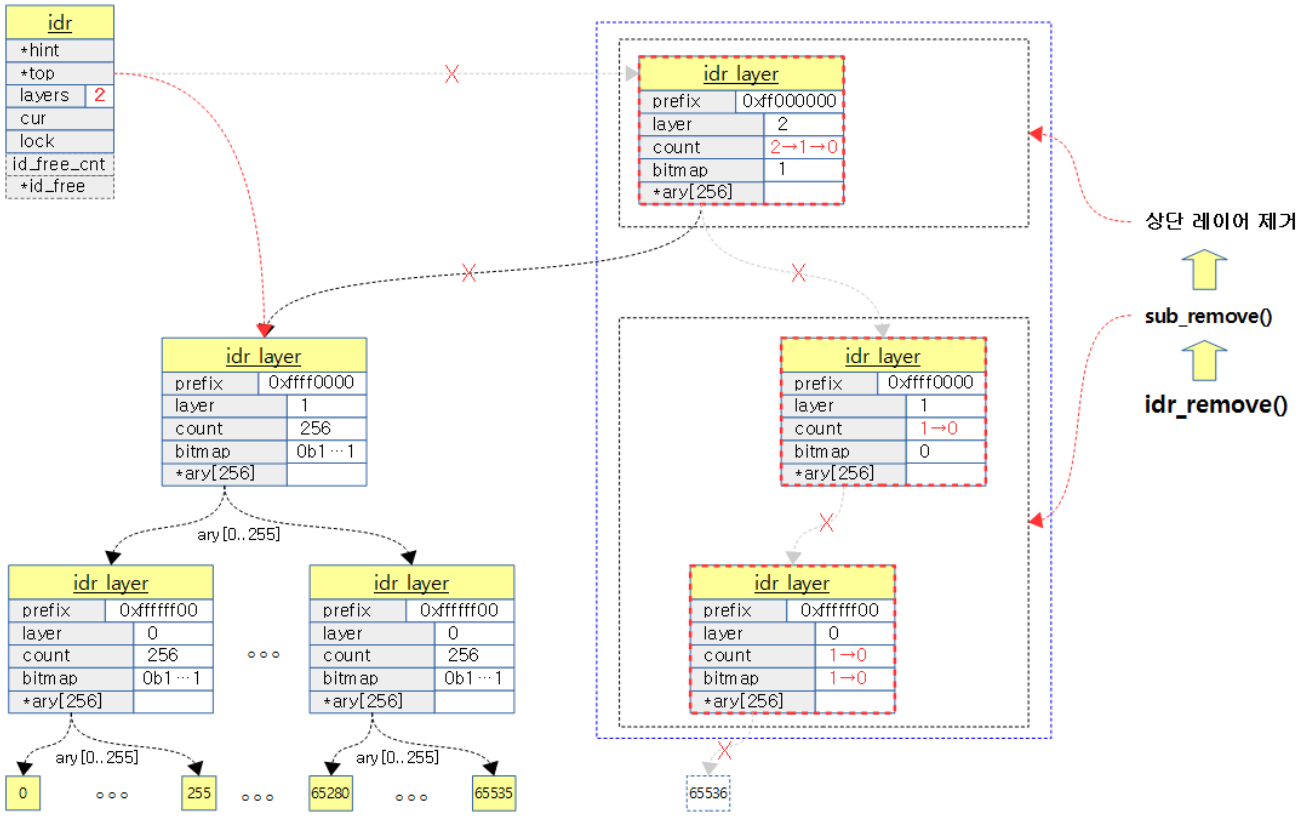
lib/idr.c

```
/**
 * idr_remove - remove the given id and free its slot
 * @idp: idr handle
 * @id: unique key
 */
void idr_remove(struct idr *idp, int id)
{
        struct idr_layer *p;
        struct idr_layer *to_free;

        if (id < 0)
                return;

        if (id > idr_max(idp->layers)) {
                idr_remove_warning(id);
                return;
        }

        sub_remove(idp, (idp->layers - 1) * IDR_BITS, id);
        if (idp->top && idp->top->count == 1 && (idp->layers > 1) &&
            idp->top->ary[0]) {
                /*
                 * Single child at leftmost slot: we can shrink the tre
e.
                 * This level is not needed anymore since when layers ar
e
                 * inserted, they are inserted at the top of the existin
g
                 * tree.
                 */
                to_free = idp->top;
                p = idp->top->ary[0];
                rcu_assign_pointer(idp->top, p);
                --idp->layers;
                to_free->count = 0;
                bitmap_clear(to_free->bitmap, 0, IDR_SIZE);
                free_layer(idp, to_free);
        }
}
EXPORT_SYMBOL(idr_remove);
```

The assigned id is removed, and the layers that were left empty during the removal process are removed. Even the depth of the layer can be reduced as needed.

- if (id < 0) return; if (id > idr_max(idp->layers)) { idr_remove_warning(id); return; }
  - If it's outside the range of identities that IDR can handle, it just slips away.
- sub_remove(idp, (idp->layers – 1) * IDR_BITS, id);
  - Without decreasing the tree depth, remove the empty layers from the connection and substitute them into the id_free.
- if (idp->top && idp->top->count == 1 && (idp->layers > 1) && idp->top->ary[0]) {
  - The top layer has a count of 1 and points to the bottom layer.
- to_free = idp->top; p = idp->top->ary[0]; rcu_assign_pointer(idp->top, p);
  - To prepare for deletion, assign the top layer to the to_free and assign the bottom layer to the top layer.
- –idp->layers; to_free->count = 0; bitmap_clear(to_free->bitmap, 0, IDR_SIZE); free_layer(idp, to_free);
  - Reduce the number of layers (tree depth), clear the count of the layers you want to delete, and the bitmap, and then turn them off.

The following diagram shows how the deletion of ID 3 from the layer in step 65536 led to the deletion of the layers and the reduction of the tree depth.



(http://jake.dothome.co.kr/wp-content/uploads/2016/08/idr_remove-1a.png)

sub_remove()

lib/idr.c

```
01 static void sub_remove(struct idr *idp, int shift, int id)
02 {
03         struct idr_layer *p = idp->top;
04         struct idr_layer **pa[MAX_IDR_LEVEL + 1];
05         struct idr_layer ***paa = &pa[0];
06         struct idr_layer *to_free;
07         int n;
08
09         *paa = NULL;
10         *++paa = &idp->top;
11
12         while ((shift > 0) && p) {
13                 n = (id >> shift) & IDR_MASK;
14                 __clear_bit(n, p->bitmap);
15                 *++paa = &p->ary[n];
16                 p = p->ary[n];
17                 shift -= IDR_BITS;
18         }
19         n = id & IDR_MASK;
20         if (likely(p != NULL && test_bit(n, p->bitmap))) {
21                 __clear_bit(n, p->bitmap);
22                 RCU_INIT_POINTER(p->ary[n], NULL);
23                 to_free = NULL;
24                 while(*paa && ! --((**paa)->count)){
25                         if (to_free)
26                                 free_layer(idp, to_free);
```

```
                        to_free = **paa;
                        **paa-- = NULL;
                }
                if (!*paa)
                        idp->layers = 0;
                if (to_free)
                        free_layer(idp, to_free);
        } else
                idr_remove_warning(id);
}
```

Without decreasing the tree depth, remove the empty layers from the ID that are to be deleted from the connection and deallocate them.

- struct idr_layer ***paa = &pa[0]; *paa = NULL; *++paa = &idp->top;
  - Assign null to pa[0] and put the top layer in pa[1].
- while ((shift > 0) && p) { n = (id >> shift) & IDR_MASK; __clear_bit(n, p->bitmap); *++paa = &p->ary[n]; p = p->ary[n]; shift -= IDR_BITS; }
  - Scroll down to the lower leaf layers, store each layer in pa[] and clear the associated bits in the bitmap.
- n = id & IDR_MASK;
  - Bit position in the lower leaf layer
- if (likely(p != NULL && test_bit(n, p->bitmap))) { __clear_bit(n, p->bitmap); RCU_INIT_POINTER(p->ary[n], NULL); to_free = NULL;
  - There is a high probability that if the bitmap of the leaf layer is set, the bits will be cleared and ary[n] will also be nulled using rcu.
- while(*paa && ! –((**paa)->count)){ if (to_free) free_layer(idp, to_free); to_free = **paa; **paa– = NULL; }
  - The layers stored in the pa[] array are looped backwards, decreasing the count of those layers, and if it is 0, it will be released if there is a layer specified in the to_free. Then, with the current layer in the to_free, it stores nulls in pa[] and specifies the next reduced pa[].
- if (!*paa) idp->layers = 0;
  - In the last case, assign 0 to idp->layers to indicate that there are no sublayers.
- if (to_free) free_layer(idp, to_free);
  - to_free Turn off the layer.

```
static inline void free_layer(struct idr *idr, struct idr_layer *p)
{
        if (idr->hint == p)
                RCU_INIT_POINTER(idr->hint, NULL);
        call_rcu(&p->rcu_head, idr_layer_rcu_free);
}
```

If idr->hint points to p to delete, assign null to the hint and then call the idr_layer_rcu_free function using rcu to free the layer.

idr_layer_rcu_free()

lib/idr.c

```
static void idr_layer_rcu_free(struct rcu_head *head)
```

```
2  {
3          struct idr_layer *layer;
4
5          layer = container_of(head, struct idr_layer, rcu_head);
6          kmem_cache_free(idr_layer_cache, layer);
7  }
```

Cancel the requested idr_layer.

Erasure IDR

idr_destroy()

lib/idr.c

```
01 /**
02  * idr_destroy - release all cached layers within an idr tree
03  * @idp: idr handle
04  *
05  * Free all id mappings and all idp_layers.  After this function, @idp i
   s
06  * completely unused and can be freed / recycled.  The caller is
07  * responsible for ensuring that no one else accesses @idp during or aft
   er
08  * idr_destroy().
09  *
10  * A typical clean-up sequence for objects stored in an idr tree will us
   e
11  * idr_for_each() to free all objects, if necessary, then idr_destroy()
   to
12  * free up the id mappings and cached idr_layers.
13  */
14 void idr_destroy(struct idr *idp)
15 {
16         __idr_remove_all(idp);
17
18         while (idp->id_free_cnt) {
19                 struct idr_layer *p = get_from_free_list(idp);
20                 kmem_cache_free(idr_layer_cache, p);
21         }
22 }
23 EXPORT_SYMBOL(idr_destroy);
```

Deletes all IDR layers and unloads the layers waiting to be assigned in the id_free list.

__idr_remove_all()

lib/idr.c

```
01 static void __idr_remove_all(struct idr *idp)
02 {
03         int n, id, max;
04         int bt_mask;
05         struct idr_layer *p;
06         struct idr_layer *pa[MAX_IDR_LEVEL + 1];
07         struct idr_layer **paa = &pa[0];
08
09         n = idp->layers * IDR_BITS;
10         *paa = idp->top;
11         RCU_INIT_POINTER(idp->top, NULL);
```

```
        max = idr_max(idp->layers);

        id = 0;
        while (id >= 0 && id <= max) {
                p = *paa;
                while (n > IDR_BITS && p) {
                        n -= IDR_BITS;
                        p = p->ary[(id >> n) & IDR_MASK];
                        *++paa = p;
                }

                bt_mask = id;
                id += 1 << n;
                /* Get the highest bit that the above add changed from 0
->1. */
                while (n < fls(id ^ bt_mask)) {
                        if (*paa)
                                free_layer(idp, *paa);
                        n += IDR_BITS;
                        --paa;
                }
        }
        idp->layers = 0;
}
```

Decommission all IDR layers.

- n = idp->layers * IDR_BITS;
  - Maximum number of bits to process
  - e.g. if layer=3 then n=24
- *paa = idp->top;
  - Assign the top layer to pa[0].
- RCU_INIT_POINTER(idp->top, NULL);
  - Assign null to idp->top to indicate that no idr_layer have been registered.
- max = idr_max(idp->layers);
  - Maximum Manageable ID Values
- id = 0; while (id >= 0 && id <= max) { p = *paa; while (n > IDR_BITS && p) { n -= IDR_BITS; p = p->ary[(id >> n) & IDR_MASK]; *++paa = p; }
  - If it's not the last leaf layer, add a layer to the pa[] array.
- bt_mask = id; id += 1 << n;
  - Hold the id and substitute the id into the starting number of the id managed by the next layer in the transverse direction.
- while (n < fls(id ^ bt_mask)) { if (*paa) free_layer(idp, *paa); n += IDR_BITS; –paa; }
  - Unlock the last layer.

Structs and Major Constants

idr_layer

include/linux/idr.h

```
struct idr_layer {
        int                     prefix; /* the ID prefix of this idr_lay
er */
        int                     layer;  /* distance from leaf */
        struct idr_layer __rcu  *ary[1<<IDR_BITS];
```

```
        int                     count;  /* When zero, we can release it
*/
        union {
                /* A zero bit means "space here" */
                DECLARE_BITMAP(bitmap, IDR_SIZE);
                struct rcu_head         rcu_head;
        };
};
```

- prefix
  - Masks that use only bits, except for the IDs managed by each layer.
  - e.g. 32-bit system
    - 0 layer (leaf node) -> 0xffffff00
    - 1 layer -> 0xffff0000
    - 2 layer -> 0xff000000
    - 3 layer -> 0x80000000
- layer
  - Layer Number (based 0)
    - leaf node is 0
- ary[256]
  - If it's not a leaf node, it points to a lower layer, and if it's a leaf node, it's used to hold the user pointer value.
- count
  - In leaf nodes, an ID is assigned to contain the number of nodes in use, and if it is not a leaf node, it contains the number of connected child nodes.
  - If this value is 0, the layer can be released.
- bitmap
  - On leaf nodes, it is set to 1 if an ID is assigned, and to 1 if the child node is full if it is not a leaf node.
- rcu_head
  - It is used as a combination with a bitmap, and is used to delete nodes using the rcu technique.

idr

include/linux/idr.h

```
struct idr {
        struct idr_layer __rcu  *hint;  /* the last layer allocated from
*/
        struct idr_layer __rcu  *top;
        int                     layers; /* only valid w/o concurrent cha
nges */
        int                     cur;    /* current pos for cyclic alloca
tion */
        spinlock_t              lock;
        int                     id_free_cnt;
        struct idr_layer        *id_free;
};
```

- hint
  - The node to which the last ID was assigned

- top
  - Highest Node
- layers
  - It is operated by increasing or decreasing according to the maximum ID value with the operating layer (tree depth).
  - If it is 0, there are no layers and no nodes are used.
  - It can be operated from 32~0 on a 4-bit system and up to 64~0 on a 8-bit system.
  - e.g. if the maximum id is 255 layers=1
- cur
  - 1
- lock
  - Locks for managing layers
- id_free_cnt
  - It contains the number of pre-allocated idr_layer entries in the cache role to make it compatible with the existing identity allocation method.
- id_free
  - In order to comply with the existing ID allocation method, idr_layer entries that are pre-assigned to the cache role are registered in this list.

IDR_BITS

- #define IDR_BITS 8
  - This value was previously increased from 5 (for 32-bit systems) or 6 (for 64-bit systems) to 2013 in 3 kernel v9.1-rc8.
  - Note: IDR: make idr_layer larger (https://github.com/torvalds/linux/commit/050a6b47d98e2bcea909c1129111e721668aaa2c)

IDR_SIZE

- #define IDR_SIZE (1 << IDR_BITS)
  - 256

IDR_MASK

- IDR_MASK ((1 << IDR_BITS)-1)
  - 255

MAX_IDR_SHIFT

- #define MAX_IDR_SHIFT (sizeof(int) * 8 – 1)
  - Number of bits used in integers, excluding signs
  - 31 (32bit system)
  - 63 (64bit system)

MAX_IDR_BIT

- #define MAX_IDR_BIT (1U << MAX_IDR_SHIFT)
  - Maximum number of integers excluding signs
  - 2^31 (32bit system)
  - 2^63 (64bit system)

MAX_IDR_LEVEL

- #define MAX_IDR_LEVEL ((MAX_IDR_SHIFT + IDR_BITS – 1) / IDR_BITS)
  - The level at which the MAX_IDR_SHIFT value can be rounded up by IDR_BITS
  - 4 Levels (32bit System)
  - 8 Levels (64bit System)

MAX_IDR_FREE

- #define MAX_IDR_FREE (MAX_IDR_LEVEL * 2)
- 8 (32bit system)
- 16 (64bit system)

consultation

- idr – integer ID management (https://lwn.net/Articles/103209/) | LWN.net
- A simplified IDR API (https://lwn.net/Articles/536293/) | LWN.net
- idr: implement idr_alloc() and convert existing users (https://lwn.net/Articles/536019/) | LWN.net
- [Linux] IDR & IDA – integer ID 관리 (http://egloos.zum.com/studyfoss/v/5187192) | F/OSS
- Trees I: Radix trees (https://lwn.net/Articles/175432/) | LWN.net

---

email *

Website

WRITE A COMMENT

‹ Vmap (http://jake.dothome.co.kr/vmap/)

idr_init_cache() › (http://jake.dothome.co.kr/idr_init_cache/)

Munc Blog (2015 ~ 2023)