Integer ID management机制

最近研究进程间通信，遇到了idr相关的函数，为了扫清障碍，先研究了linux的idr机制。
IDR(integer ID management)是给要管理的对象分配一个唯一的ID，于是可以通过这个数字找到要管理的对象。

应用IDR机制时要包含头文件<linux/idr.h>。

```
struct idr {

    struct idr_layer *top;  //idr的top层，可以方便的理解为根节点。
    struct idr_layer *id_free; //id_free为首的形成一个链表，这个是预备队，
                                          //并没有参与到top为根的节点中去
    int layers; //当前的层数。
    int id_free_cnt;// 预备队的个数。
    spinlock_t lock;

};
```

```
struct idr_layer {

    unsigned long
    bitmap; /* A zero bit means "space here" */

    struct idr_layer
    *ary[1<<IDR_BITS];

    int count; /* When zero, we can release it */
};
```

```
IDR_BITS 在32位操作系统是5 ，64位操作系统是6，我们以32位操作系统为例。
```

本文的介绍以两层的为例。layers = 2. idr中的top指向的是当前正在工作的最高层的idr_layer，即图中的A，top的ary是个指针数组，指向 低一层的idr_layer。top层ary指针数组不一定都指向已经分配了的低一层idr_layer。也可能某个指针指 向NULL。如下图的ary[1]就指向NULL。
最后一层idr_layer 叶子层 例如B，他的指针数组ary中的元素，如果分配出去了那么指向某个结构体的地址，这个地址指向要管理的数据结构。如果没有分配出去，指针指向NULL。对于叶子层而言，判断指针数组某个元素是否指向有意义的数据结构，用位图bitmap。bitmap对应的位 是1，表示ary数组的对应元素指向某有意义的数据结构。
最后一层的bitmap的含义已经介绍，但是top层（或者层数大于2的时候，中间某层）bitmap的含义是什么呢？以两层为例，如果图中B的bitmap是0xFFFFFFFF，即每一个指针都分配出去了，那么A的bitmap的第0位置1.同样如果A的bitmap的第2位是1，表示ary[2]指向的C的bitmap是0xFFFFFFFF，即C也ary数组也分配完毕。

这部分是函数idr_mark_full来实现:

```
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);// 叶子层数字id对应的位 置1.

    /*
     * 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 (p->bitmap == IDR_FULL) {

        if (!(p = pa[++l]))             // pa[++l]记录的上一层idr_layer。
            break;

        id = id >> IDR_BITS;

        __set_bit((id & IDR_MASK), &p->bitmap); //如果由于本层满了，则上一层对应位置
1.
    } //循环检测。
}
```

介绍完负责工作的部分，下面介绍预备役。所谓预备役就是id_free指向的空闲的idr_layer。所谓空闲是指，这些idr_layer并没有投入。如果需要分配一个idr_layer，首先将id_free指向的idr_layer取出来使用，同时id_free指向下一个。即如下图所示，如果需要分配，D被取出来使用，同时id_free指针指向E，同时id_freecnt减一。

将预备役投入使用是函数alloc_layer完成的:

```
static struct idr_layer *alloc_layer(struct idr *idp)
{
    struct idr_layer *p;
    unsigned long flags;

    spin_lock_irqsave(&idp->lock, flags);

    if ((p = idp->id_free)) {

        idp->id_free = p->ary[0]; // id_free 指向D的下一位 E
        idp->id_free_cnt--;         // 预备役的个数减1
        p->ary[0] = NULL;          //D要被使用了，第0个指针不再指向E，初始化为NULL
    }

    spin_unlock_irqrestore(&idp->lock, flags);
```

```
    return(p);  // 返回D
}
```

有个问题是预备役是怎么来的?如果预备役分配光了怎么办。分配光了也没有关系，还好我们有idr_pre_get函数。

```
#if BITS_PER_LONG == 32
#define IDR_BITS 5
#define MAX_ID_SHIFT (sizeof(int)*8 - 1)               //31
#define MAX_LEVEL         (MAX_ID_SHIFT + IDR_BITS - 1) / IDR_BITS
//7
#define IDR_FREE_MAX      MAX_LEVEL + MAX_LEVEL
//14
```

坦白说，MAX_LEVEL的含义是什么，我并不清楚。为什么一次分配14个idr_layer充当预备役我并不清楚。请清楚的兄弟不吝赐教。
这个函数的含义就是我要分配14个idr_layer，充当预备役。如果中间分配失败，那么能分配几个算几个。投入预备役的函数是free_layer。比较好懂我就不解释了。

```
int idr_pre_get(struct idr *idp, gfp_t gfp_mask)
{
    while (idp->id_free_cnt < IDR_FREE_MAX) {

        struct idr_layer *new;
        new = kmem_cache_alloc(idr_layer_cache, gfp_mask);

        if (new == NULL)
            return (0);

        free_layer(idp, new);
    }

    return 1;
}
```

```
static void free_layer(struct idr *idp, struct idr_layer *p)
{
    unsigned long flags;

    /*
     * Depends on the return element being zeroed.
     */

    spin_lock_irqsave(&idp->lock, flags);

    __free_layer(idp, p);
```

```
    spin_unlock_irqrestore(&idp->lock, flags);
}
```

```
static void __free_layer(struct idr *idp, struct idr_layer *p)
{
    p->ary[0] = idp->id_free;
    idp->id_free = p;
    idp->id_free_cnt++;
}
```

从预备役机制上看，我们可以得到使用idr编程流程应该是这样的。 首先调用idr_pre_get，来分配可用的idr_layer，投入预备役，接下来调用idr_get_new， 给要管理的对象target分配一个数字id，这个过程中可能会调用alloc_layer，将预备役中的 idr_layer投入使用，用在top为根管理结构中。 终有一天，预备役也被打光了idr_get_new 函数返回-EAGAIN，告诉我们，预备役全部阵亡，于是，我们从-EAGAIN的遗言中，知道，我们需要调用 idr_pre_get来充实预备役了。

```
again:
if (idr_pre_get(&my_idr, GFP_KERNEL) == 0) {
    /* No memory, give up entirely */
}
spin_lock(&my_lock);
result = idr_get_new(&my_idr, &target, &id);
if (result == -EAGAIN) {
    sigh();
    spin_unlock(&my_lock);
    goto again;
}
```

下面：讲述如何给要管理的对象分配一个小数字作为id。 首先看知道obj的ID，如果查找obj ，即指向obj的指针。也就是说先看我们想要达到的效果，在来分析如何实现给对象分配ID。 根据ID ,来查找obj。函数idr_find实现查找功能 假如下图中C 的ary[2]指向一个管理的obj。我们来看下如何通过数字66来查找到obj。 我们以top为根的树其实是一个32叉树。如果只有一层，那么top本身指向叶子层，那么最多理32个obj，即ary数组的每个元素，指向一个obj。 但是假如说我们管理的对象超过了32个，我们就不能用一层来管理这个需要有两层结构。就像我们的示意图。 其实idr有一种比较简单的理解方式，就是它是一种32进制的数，满32，向前进一位。 我们还是从示意图讲起。我们寻找66指向的obj。首先判断66是否超过了当前层数所能管理最多obj。 当前我们是两层结构，top指向32叉树的根，top下面管理32个叶子层的idr_layer。上面一讲提到了，叶子层idr_layer的ary数组元素是用来指向目标obj的。那么两层总共可以管理32*32=1024个obj。同样道理三层可以最多管理*32*32*32=32K 个obj。 要想找到obj的指针，必须根据ID，一路寻找的叶子层。66/32 = 2,所以从top--->top->ary[2], 我们就找到了叶子节点C。66|IDR_MASK = 2，所以C的ary[2]指向管理的obj。

用前面的32进制方法理解就是66 = *32+2，所以，top->ary[2]->ary[2]指向obj。 同样我们可以求ID是27对应的obj 27=0*32+27，所以top->ary[0]->ary[27]指向obj。

小结: 通过上面的描述，我们也看到了，我们就是要建立一个32叉树，来管理obj。通过ID，可以一层层定位到叶子层，叶子层的指针指向的就是我们要管理的obj。 需要指出的是32叉树，不一定每个分支都分配好了idr_layer，用到了再分配，防止浪费，比如示意图中，并没有用到32~63，我们看到top->ary[1]为NULL。如有需要分配34了，那没办法，会在分配过程中分配个idr_layer,top->ary[1]指向分配的idr_layer。

```
void *idr_find(struct idr *idp, int id)
{
    int n;
    struct idr_layer *p;
    n = idp->layers * IDR_BITS;
    p = idp->top;

    /* Mask off upper bits we don't use for the search. */
    id &= MAX_ID_MASK;
    if (id >= (1 << n))
        return NULL;

    while (n > 0 && p) {
        n -= IDR_BITS;
        p = p->ary[(id >> n) & IDR_MASK];
    }

    return((void *)p);
}
```

下面分析如果给一个obj分配个ID。 提供两个函数给obj分配ID

```
int idr_get_new(struct idr *idp, void *ptr, int *id)
int idr_get_new_above(struct idr *idp, void *ptr, int starting_id, int *id)
```

参数说明:

idp: 不说了，管理结构idr的指针，对应示意图中最左面的那个结构。 **ptr:** 指向要管理的结构的指针，我们的任务就是给它分配个小数字，作为他的身份证。成功之后,我们可以拿着这个ID，直接找到ptr。 **id:** 输出参数，将分配的数字存入id。

这两个函数其中idr_get_new比较乖，比较好说话，随便给他分配一个没人用的id就可以，他他不挑不捡。第二个函数idr_get_new_above有点难说话，要求挺多，他有个参数starting_id，要求分配不小于starting_id的一个数字作为id。 两个函数都是调用了idr_get_new_above_int，区别是idr_get_new将starting_id填成了0.表示随便给分配个大于0的没被别人用的id就行。 -EAGAIN的意思上面一讲提到过，这个是预备役全体阵亡的遗言，没有空闲的idr_layer用来分配了，所以失败了，如果用户非常需要给ptr分配个id，那么请先分配点预备役，即调用idr_pre_get。 -ENOSPC的含义是你小子要的id太大了，超过了MAX_ID_BIT，即2^31，idr说，我是管理小数字的结构，拜托不要那这么大的数字骚扰我。

```
if ((id >= MAX_ID_BIT) || (id < 0))
    return -3;           //  sub_alloc函数中的语句
```

```
int idr_get_new(struct idr *idp, void *ptr, int *id)
{
    int rv;
    rv = idr_get_new_above_int(idp, ptr, 0);
    /*
    * This is a cheap hack until the IDR code can be fixed to
    * return proper error values.
    */

    if (rv < 0) {
        if (rv == -1)
            return -EAGAIN;
        else /* Will be -3 */
            return -ENOSPC;
    }

    *id = rv;
    return 0;
}
```

酝酿了半天，可以聊聊idr_get_new_above_int这个了。

idr_get_empty_slot函数是分配个大于starting_id的数字作为ptr的ID。如果分配成功，id>=0,将叶子节点id对应的ary数组的元素赋值为 ptr。同时将叶子层的count++，表示又分配出去一个。将叶子层的位图bitmap对应槽位置1的工作是idr_mark_full完成。如果叶子层全满了，则通知叶子层的父亲对应槽位置1，依次传递。

```
static int idr_get_new_above_int(struct idr *idp, void *ptr, int starting_id)
{

    struct idr_layer *pa[MAX_LEVEL];
    int id;
    id = idr_get_empty_slot(idp, starting_id, pa);

    if (id >= 0) {
        /*
        * Successfully found an empty slot.  Install the user
        * pointer and mark the slot full.
        */
        pa[0]->ary[id & IDR_MASK] = (struct idr_layer *)ptr;
        pa[0]->count++;
        idr_mark_full(pa, id);
    }
    return id;

}
```

OK，到了idr_get_empty_slot。这个函数是干重活的函数。需要仔细研读代码。这个函数不举例子很难描述清楚，举例子又显得特别琐碎，很头疼。建议读者从0开始分配一直分配到32需要分层，就可以理解代码的含

义。 先讲初始化：

```
#define IDR_INIT(name)                          \
{                                       \
    .top        = NULL,                     \
    .id_free    = NULL,                     \
    .layers     = 0,                    \
    .id_free_cnt    = 0,                    \
    .lock        = __SPIN_LOCK_UNLOCKED(name.lock),    \
}
```

top等于NULL 表示我的32叉树还没建立起来，id_free =NULL，id_free_cnt=0表示不好意思，我的预备役也为空，没法为您分配idr_layer。这是最初的状态，32叉树连个根都没有，整个idr处于一穷二白的状态。

```
p = idp->top;
layers = idp->layers;

if (unlikely(!p)) {

    if (!(p = alloc_layer(idp)))
        return -1;
    layers = 1;

}
```

idr_get_empty_slot这个部分，表示如果ｉｄｒ的３２叉树连个根都没有，我需要分配一个idr_layer来当根。如果alloc_layer失败，表示预备役空了，惨了，只能返回失败，告诉调用者，预备役没了，请填充预备役。一般是可以分配的。

这个循环体的含义是，用户这个搞得这个starting_id太大了，或者低的id分配出去了，只能给用户分配个大的id。如果这个id大于了当前层数所能管理的最高ID，我们需要加一层了。

以上面的示意图为例，我们当前有两层结构，最多能管理32*32=1K个，我们能分配的最大id就是1023，如果用户要求我们分配大于等于1500的id，那么我们目前的两层结构是无法满足需要的，所以我们需要加一层。首先将layer++，表示我们的32叉树升级了，多了一层，从预备役分配出一个idr_layer，让新分配的new当根。p指针指向根。

如果分配的id不够大，不需要分层，那么这个while就不执行了，直接跳到sub_alloc函数。

```
while ((layers < (MAX_LEVEL - 1)) && (id >= (1 << (layers*IDR_BITS)))) {

    layers++;
    if (!p->count)//这个地方是应对特殊情况，比如0~31都没有分配，第一层还没有，用户
        continue; //上来要分配32或46这样明显是两层才能完成的结构

    if (!(new = alloc_layer(idp))) {
        /*
```

```
            * 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->bitmap = new->count = 0;
                __free_layer(idp, new);
            }

            spin_unlock_irqrestore(&idp->lock, flags);
            return -1;

        }

        new->ary[0] = p;
        new->count = 1;
        if (p->bitmap == IDR_FULL)
        __set_bit(0, &new->bitmap);
        p = new;
    }

    idp->top = p;
    idp->layers = layers;
    v = sub_alloc(idp, &id, pa);
    if (v == -2)
        goto build_up;
```

sub_alloc函数。

还是以示意图为例讲述。我们是两层的结构，p是32叉树的根节点top

如果用户要分配大于等于66的id，66=2*32+2，首先找到了我们要找的66是位于top->ary[2],我们需要确认 根的ary[2]这个分支是否还能分配。如果p->ary[2]对应的idr_layer 所有的槽位都分配出去了，客满，新的顾客无法入住，我们就不必白费劲去ary[2]这个分支去分配了。判断的办法就是m = find_next_bit(&bm, IDR_SIZE, n); 这个函数很可爱，就是说我要找大于2 的所有分支，寻找第一个没有客满的分支。通过top层或者中间层bitmap的含义，如果某个分支全部客满，则在对应bitmap位置1 ，表示，不要去这个分支找了，找也白找。

然后一层层往下找，知道找到叶子层，在叶子层查找大于等于2的id。 各种情况我就不分析了，大家可以自己尝试分配一下:

1. 从0开始，分配，累加到33，差不多就可以理解idr_get_new这种情况的分配流程
2. 不按常理出牌，乱分配，假如我第一个就要分配 大于37的，第二次就要分配大于1500的，之类的， 在走一遍流程，就可以理解相关的代码。

```
while (1) {
    /*
     * We run around this while until we reach the leaf node...
     */
```

```
        n = (id >> (IDR_BITS*l)) & IDR_MASK;
        bm = ~p->bitmap;
        m = find_next_bit(&bm, 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 (!(p = pa[l])) {
                *starting_id = id;
                return -2;
            }
            /* 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;
        }
        if (m != n) {
            sh = IDR_BITS*l;
            id = ((id >> sh) ^ n ^ m) << sh;
        }
        if ((id >= MAX_ID_BIT) || (id < 0))
            return -3;
        if (l == 0)
            break;
        /*
         * Create the layer below if it is missing.
         */
        if (!p->ary[m]) {
            if (!(new = alloc_layer(idp)))
                return -1;
            p->ary[m] = new;
            p->count++;
        }
        pa[l--] = p;
        p = p->ary[m];
    }
```

参考文献： IDR-integer ID management