Estructuras de datos en el Kernel

Miguel Higuera Romero
Alejandro Nicolás Ibarra Loik

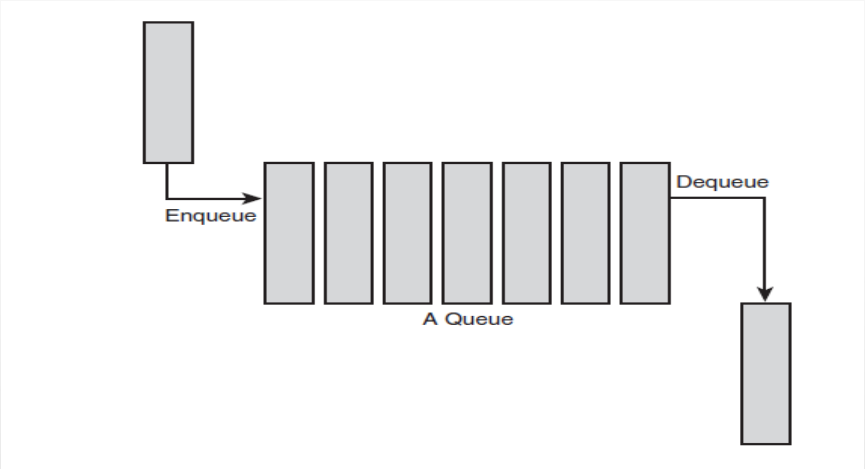
Introducción

- En el kernel de Linux se utilizan colecciones de datos. Para el manejo de las mismas se utilizan estructuras de datos.
- Explicaremos los tipos de estructuras de datos básicas que hay, cómo se implementan en el kernel, las estructuras internas que utilizan y un breve ejemplo de uso

- Dependiendo del trato que se le vaya a dar a la colección:
  - Listas enlazadas
  - Colas
  - Mapas
  - Árboles binarios

Colas -Visión general

- Es la forma más fácil de implementar el patrón productor-consumidor.
- El productor coloca datos en la cola y el consumidor toma los datos de la cola.
- Por esta razón, las colas son también llamados FIFOs (first-in, first-out).



Colas en el kernel

- La implementación genérica de una cola en el kernel de linux se llama “kfifo”
- Provee de las funcionalidades básicas para:
  - Crear la cola
  - Añadir elementos a la cola
  - Extraer elementos de la cola
  - Obtener número de elementos de la cola
  - Eliminar la cola

Colas en el kernel – Estructura kfifo

- En linux, la estructura kfifo se encuentra declarada en <linux/kfifo.h>
- Mantiene dos off-set en la cola: un "in offset" y un "out offset"
- Indican los extremos de encolado y desencolado de la cola respectivamente.

```
struct __kfifo {
        unsigned int    in;
        unsigned int    out;
        unsigned int    mask;
        unsigned int    esize;
        void            *data;
};
```

Colas en el kernel – Estructura kfifo

- Operaciones en kfifo

  - Creación de la cola:

```
int kfifo_alloc(struct kfifo *fifo,
                unsigned int size, gfp_t
                gfp_mask);
```

`fifo` – puntero a la estructura kfifo

`size` – tamaño total de la kfifo (**`PAGE_SIZE`**)

`gfp_mask` – flag de reseva de memoria (**`GFP_KERNEL`**)

Colas en el kernel – Estructura kfifo

- Operaciones en kfifo
  - Añadir elemento a la cola:

```
unsigned int kfifo_in(struct kfifo
                      *fifo, const void *from,
                      unsigned int len);
```

Copia **`len`** bytes comenzando en **`from`** dentro de la cola **`fifo`**

Retorna el numero de bytes que se han añadido a la cola

Colas en el kernel – Estructura kfifo

- Operaciones en kfifo

  - Extraer elemento de la cola:

```
unsigned int kfifo_out(struct kfifo
                       *fifo, void *to, unsigned
                       int len);
```

Copia (como mucho) **len** bytes de la cola a **to**

Retorna el número de elementos quitados de la cola

Colas en el kernel – Estructura kfifo

- Operaciones en kfifo

  - Extraer elemento de la cola:

```
unsigned int kfifo_out_peek(struct kfifo
                            *fifo, void *to, unsigned
                            int len, unsigned inr offset);
```

Funcionamiento parecido al de kfifo_out solo que no remueve el elemento de la cola.

El parámetro **offset** indica el índice dentro de la cola

Colas en el kernel – Estructura kfifo

- Operaciones en kfifo

  - Devolver tamaño de la cola:

```
static inline unsigned int kfifo_size(struct kfifo *fifo);
static inline unsigned int kfifo_len(struct kfifo *fifo);
static inline unsigned int kfifo_avail(struct kfifo *fifo);
static inline int kfifo_is_empty(struct kfifo *fifo);
static inline int kfifo_is_full(struct kfifo *fifo);
```

  - Eliminar la cola:

```
void kfifo_free(struct kfifo *fifo);
```

Colas en el kernel – Ejemplo de uso

```
int test(void)
{

struct kfifo fifo;
int ret, tam;
char *buf;

/* Inicializo la cola y compruebo errores */
ret = kfifo_alloc(&fifo, FIFO_SIZE, GFP_KERNEL);
if (ret) {
   printk(KERN_ERR "error kfifo_alloc\n");
   return ret;
}

/* Introduzco la cadena Hello en la cola */
kfifo_in(&fifo, "Hello", 5);

/* Introduzco la cadena LIN de la cola */
kfifo_in(&fifo, "LIN", 3);
```

Colas en el kernel – Ejemplo de uso

```
/* Extraigo la cadena Hello de la cola */
tam = kfifo_out(&fifo, buf, 5);
printk(KERN_INFO "%s", buf);

/* Devuelvo la cadena LIN sin extraerla de la cola */
if (kfifo_peek(&test, buf))
   printk(KERN_INFO " %s\n", buf);

/* Elimino la cola */
kfifo_free(&test);

return 0;
}
```

Colas en el kernel – Ejemplo de uso

```
/* Extraigo la cadena Hello de la cola */
tam = kfifo_out(&fifo, buf, 5);
printk(KERN_INFO "%s", buf);

/* Devuelvo la cadena LIN sin extraerla de la cola */
if (kfifo_peek(&test, buf))
   printk(KERN_INFO " %s\n", buf);

/* Elimino la cola */
kfifo_free(&test);

return 0;
}
```

Mapas – Visión general

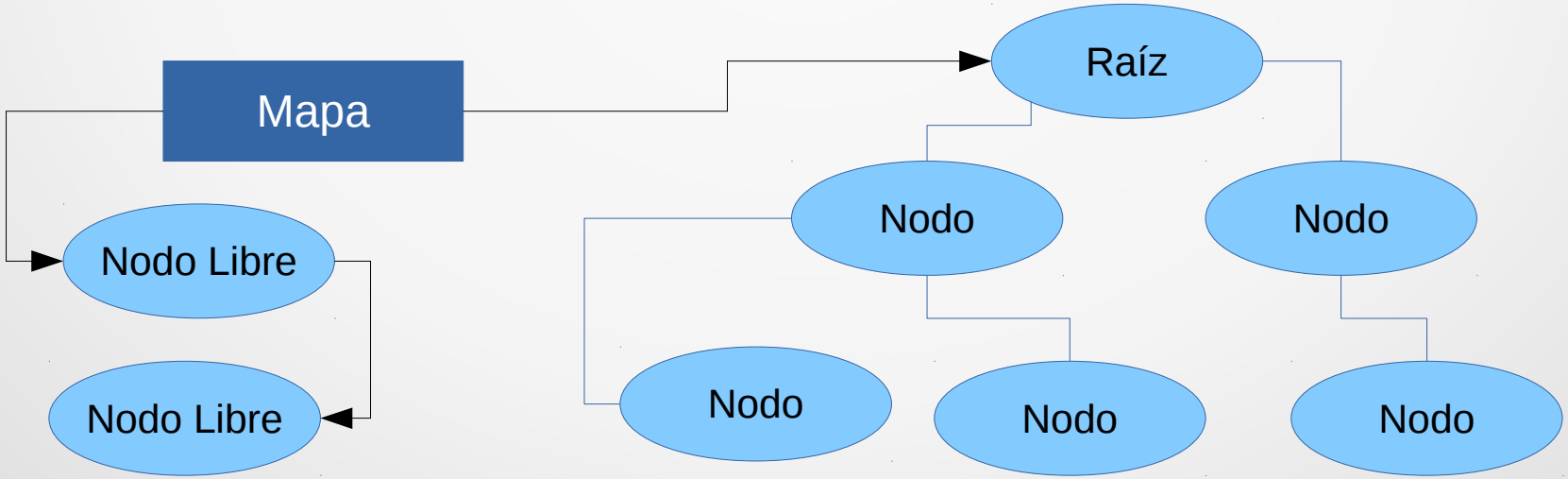
- Son colecciones de pares (clave, valor) en los que a través de una clave accedemos a un valor.
- Suelen estar implementados con *tablas Hash* o con *árboles binarios* auto-balanceados.
- Los mapas deben cumplir con las operaciones:
  - añadir(clave, valor)
  - eliminar(clave)
  - valor = consultar(clave)

Mapas en el Kernel

- En el kernel se proporciona una estructura de datos simple y eficiente.
- No son de uso general. El cometido no es otro que mapear un UID a una estructura de datos en el kernel.
- Se utilizan árboles binarios.

Mapas en el Kernel

- En el kernel se proporciona una estructura de datos simple y eficiente.
- No son de uso general. El cometido no es otro que mapear un UID a una estructura de datos en el kernel.
- Se utilizan árboles binarios.



Mapas en el Kernel – Estructura idr

- En Linux se utiliza la estructura ***idr*** para implementar los mapas. <linux/idr.h>
- Con ello se pueden mapear UIDs del espacio de usuario con sus estructuras de datos en el kernel.
  - Por ejemplo los IDs de los temporizadores POSIX.
    - int timer_create(clockid_t clockid, struct sigevent *sevp, timer_t *timerid);
- La función de inserción se encarga de reservar memoria, generar el UID y añadir el par (clave,valor) en el mapa.

Mapas en el Kernel – Estructura idr

Mapa

```
struct idr {
    struct idr_layer *hint;
    struct idr_layer *top;
    int layers;
    int cur;
    spinlock_t lock;
    int id_free_cnt;
    struct idr_layer *id_free;
};
```

Nodo

```
struct idr_layer {
  int prefix;    // UID
  int layer;     // Distancia hojas
  struct idr_layer *ary[1<<IDR_BITS];
  int count;
  union {
    DECLARE_BITMAP(bitmap, IDR_SIZE);
    struct rcu_head rcu_head;
  };
};
```

Mapas en el Kernel – Funciones idr

- Inicializar un mapa idr:
  - `void idr_init(struct idr *idp);`
- Destruir un mapa idr (Debe estar vacío):
  - `void idr_destroy(struct idr *idp);`
- Introducir nuevo dato:
  - `int idr_pre_get(struct idr *idp, gfp_t gfp_mask);`
  - `int idr_get_new(struct idr *idp, void *ptr, int *id);`
- Buscar en el mapa:
  - `void *idr_find(struct idr *idp, int id);`
- Eliminar del mapa:
  - `void idr_remove(struct idr *idp, int id);`
  - `void idr_remove_all(struct idr *idp);`

Mapas en el Kernel – Ejemplo de uso

```
int id;
struct idr idr_huh;
struct my_struct *ptr = ...;
/** Inicialización **/
idr_init(&idr_huh);
/** Inserción **/
do{
     if(!idr_pre_get(&idr_huh, GFP_KERNEL))
          return -ENOSPC;
     ret = idr_get_new(&idr_huh, (void *)ptr, &id);
}while(ret == -EAGAIN);
```

Mapas en el Kernel – Ejemplo de uso

```
int id;

struct idr idr_huh;

struct my_struct *ptr = ...;

/** Inicialización **/

idr_init(&idr_huh);

/** Inserción **/

do{

     if(!idr_pre_get(&idr_huh, GFP_KERNEL))

          return -ENOSPC;

     ret = idr_get_new(&idr_huh, (void *)ptr, &id);

}while(ret == -EAGAIN);
```

Se encarga de generar nodos libres para insertarlos al mapa

Mapas en el Kernel – Ejemplo de uso

```
int id;

struct idr idr_huh;

struct my_struct *ptr = ...;

/** Inicialización **/

idr_init(&idr_huh);

/** Inserción **/

do{

     if(!idr_pre_get(&idr_huh, GFP_KERNEL))

          return -ENOSPC;

     ret = idr_get_new(&idr_huh, (void *)ptr, &id);

}while(ret == -EAGAIN);
```

Se encarga de generar nodos libres para insertarlos al mapa

Genera el UID y añade (clave, valor) al mapa

Mapas en el Kernel – Ejemplo de uso

```
/** Consulta **/
ptr = (struct my_struct *)idr_find(&idr_huh, id);
if(!ptr)
     return -EINVAL;

/** Eliminación de id **/
idr_remove(&idr_huh, id);

/** Eliminación de mapa **/
idr_destroy(&idr_huh);
```

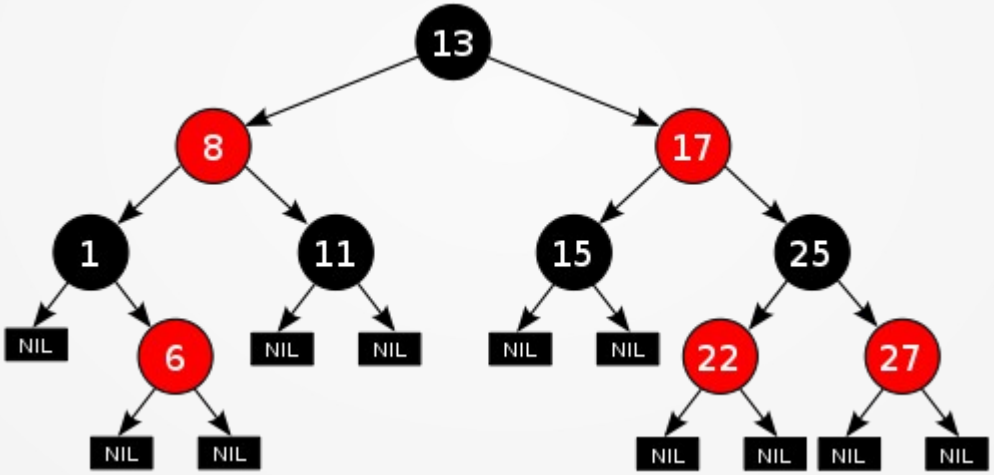
Mapas en el Kernel – Ejemplo de uso

```
/** Consulta **/
ptr = (struct my_struct *)idr_find(&idr_huh, id);
if(!ptr)
     return -EINVAL;

/** Eliminación de id **/
idr_remove(&idr_huh, id);

/** Eliminación de mapa **/
idr_destroy(&idr_huh);
```

Hay que vaciar el mapa antes!!!
idr_remove_all(&idr_huh)

Árboles RN – Visión general

Los árboles RN son árboles binarios de búsqueda auto-balanceados.



Árboles RN – Visión general

Un árbol RN debe cumplir con lo siguiente:

- Un nodo es rojo o negro
- Los nodos hoja son negros y no contienen datos
- Todos los nodos no-hoja tienen dos hijos
- Si un nodo es rojo sus dos hijos son negros
- El camino desde un nodo a una de sus hojas contiene el mismo número de nodos negros como el camino más corto a cualquiera de sus otras hojas.

- Inserción y eliminación deben cumplir estas características.
- Más información → DA / TAIS

Árboles RN en el Kernel - rbtree

- La implementación en Linux de los árboles RN está en *<linux/rbtree.h>* y cumple con las características mencionadas de árboles rojinegros.

- Sin embargo no se ofrecen las funciones de inserción y búsqueda en la implementación de *rbtree*.
  - El programador debe implementarlas y definir sus operadores de comparación.
  - Existen funciones de ayuda de *rbtree*

Árboles RN en el Kernel - rbtree

```
struct rb_root {
        struct rb_node *rb_node;
};
struct rb_node {
        unsigned long  __rb_parent_color;
        struct rb_node *rb_right;
        struct rb_node *rb_left;
};
```

- Creación de un nuevo árbol:

```
struct rb_root root = RB_ROOT;
```

Árboles RN – Ejemplo de uso

- Búsqueda de número entero en el árbol:

```
struct my_struct * rb_search_num( struct rb_root * root , int target ){
        struct rb_node * n = root->rb_node;
        struct my_struct * ans;
        while( n ){
                //Get the parent struct to obtain the data for comparison
                ans = rb_entry( n , struct my_struct , __nodo );
                if( target < ans->__num )
                        n = n->rb_left;
                else if( target > ans->__num )
                        n = n->rb_right;
                else
                        return ans;
        }
        return NULL;
}
```

```
struct my_struct {
        struct rb_node __nodo
        int __num
}
```

Árboles RN – Ejemplo de uso

- Inserción de estructura en el árbol:

```
struct my_struct * rb_insert_node( struct rb_root * root , int target , struct rb_node * source ){
        struct rb_node **p = &root->rb_node;
        struct rb_node *parent = NULL;
        struct my_struct * ans;
        while( *p ){
                parent = *p;
                ans = rb_entry( parent , struct my_struct , __nodo );
                if( target < ans->__num )
                        p = &(*p)->rb_left;
                else if( target > ans->__num )
                        p = &(*p)->rb_right;
                else
                        return ans;
        }
        rb_link_node( source , parent , p );
        rb_insert_color( source , root );
        return NULL;
}
```

```
struct my_struct {
        struct rb_node __nodo
        int __num
}
```

Árboles RN – Ejemplo de uso

- Inserción de estructura en el árbol:

```
struct my_struct * rb_insert_node( struct rb_root * root , int target , struct rb_node * source ){
        struct rb_node **p = &root->rb_node;
        struct rb_node *parent = NULL;
        struct my_struct * ans;
        while( *p ){
                parent = *p;
                ans = rb_entry( parent , struct my_struct , __nodo );
                if( target < ans->__num )
                        p = &(*p)->rb_left;
                else if( target > ans->__num )
                        p = &(*p)->rb_right;
                else
                        return ans;
        }
        rb_link_node( source , parent , p );
        rb_insert_color( source , root );
        return NULL;
}
```

```
struct my_struct {
        struct rb_node __nodo
        int __num
}
```

Insertamos el nuevo nodo como una hoja roja!

Árboles RN – Ejemplo de uso

- Inserción de estructura en el árbol:

```
struct my_struct * rb_insert_node( struct rb_root * root , int target , struct rb_node * source ){
        struct rb_node **p = &root->rb_node;
        struct rb_node *parent = NULL;
        struct my_struct * ans;
        while( *p ){
                parent = *p;
                ans = rb_entry( parent , struct my_struct , __nodo );
                if( target < ans->__num )
                        p = &(*p)->rb_left;
                else if( target > ans->__num )
                        p = &(*p)->rb_right;
                else
                        return ans;
        }
        rb_link_node( source , parent , p );
        rb_insert_color( source , root );
        return NULL;
}
```

```
struct my_struct {
        struct rb_node __nodo
        int __num
}
```

Insertamos el nuevo nodo como una hoja roja!

Y rebalanceamos el árbol acabando la inserción

¿Qué estructura de datos usar?

Lo que queremos hacer	Estructura de datos adecuada
Iteración sobre datos	Listas enlazadas
Patrón productor consumidor	Colas (FIFO)
Mapear un UID a un objeto	Mapas
Guardar una cantidad grande de datos y buscar sobre ella eficientemente	Árboles RojiNegros

Bibliografía

- Linux Kernel Development – Robert Love
- http://lxr.free-electrons.com/
- EDA, DA / TAIS