Path.c

1. Introducción

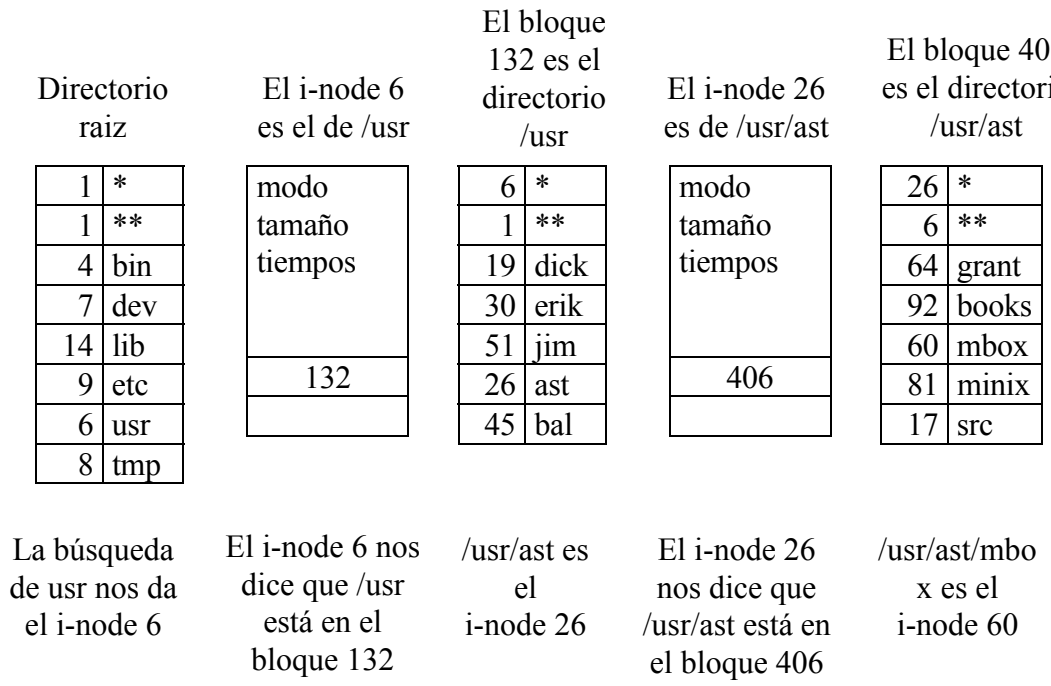
Antes de entrar en cómo funciona el path.c, veamos algunas definiciones y aclaraciones para entender mejor el funcionamiento de este fichero.

Un **directorio** es un fichero que contiene entradas de 16 bytes. Los dos primeros bytes forman un número de i-node de 16 bits y los 14 bytes restantes son el nombre del archivo.

Todo directorio tiene entradas '.' y '..', que se colocan cuando se crea el mismo. La entrada '.' tiene asociado el número del i-node del directorio actual y la '..' el número del i-node del directorio padre del actual.

La única complicación que puede existir es la que se presenta cuando se encuentra un sistema de archivo montado. Eso se explicará en el fichero mount.c.

Para ver cómo se encadena un path, veamos el siguiente ejemplo:
/usr/ast/mbox



2. Conversión de un path en su correspondiente i-node

Muchas llamadas al sistema, como OPEN, UNLINK, MOUNT, etc., requieren el nombre del path (p.e. nombres de fichero) como parámetro. La mayoría de estas llamadas deben encontrar el i-node para dichos ficheros antes de que puedan trabajar con ellos. Cómo se realiza esta conversión es lo que vamos a ver.

La comprobación de nombres de path se hace en el fichero path.c. El primer procedimiento **eat_path** coge un nombre de path, lo analiza, y devuelve un puntero al i-node a cargar en memoria. Esto se hace llamando a **last_dir**, que obtiene el i-node del último directorio del path y luego llama a **advance** para obtener el último componente del path. Si la búsqueda falla, por ejemplo porque uno de los directorios a lo largo del path no existe, o existe pero está protegido contra accesos, entonces, se devuelve un *NIL_INODE*. Los nombres del path pueden ser absolutos o relativos y pueden tener arbitrariamente muchos componentes separados por barras. Esto se trata en **last_dir**, que empieza examinando el primer caracter del path para ver si es absoluto o relativo, esto es, empezar a buscar en el directorio raíz, o en el directorio de trabajo.

A estas alturas, **last_dir** tiene el nombre del path y un puntero al i-node del directorio donde buscar la primera componente. Entra en una bucle analizando el nombre del path componente a componente y cuando llega al final devuelve un puntero al último directorio.

Get_name es un procedimiento de utilidad que extrae componentes de una ristra. **Advance** toma como parámetros un puntero al directorio y una ristra y busca la ristra en el directorio. Si lo encuentra, devuelve un puntero a su i-node. Los detalles de paso a sistemas de ficheros montados se manejan también en este punto.

Aunque **advance** controla la búsqueda de la ristra, la comparación real de la ristra con las entradas del directorio se hace en **search_dir** que es el único lugar en el sistema del ficheros donde se examinan realmente los ficheros del directorio. Contiene dos bucles anidados, uno para hacer la búsqueda de bloques en un directorio, y otro para las entradas en un bloque. **Search_dir** también se usa para crear y borrar nombres de los directorios. La figura siguiente muestra las relaciones entre algunos de los procedimientos usados.

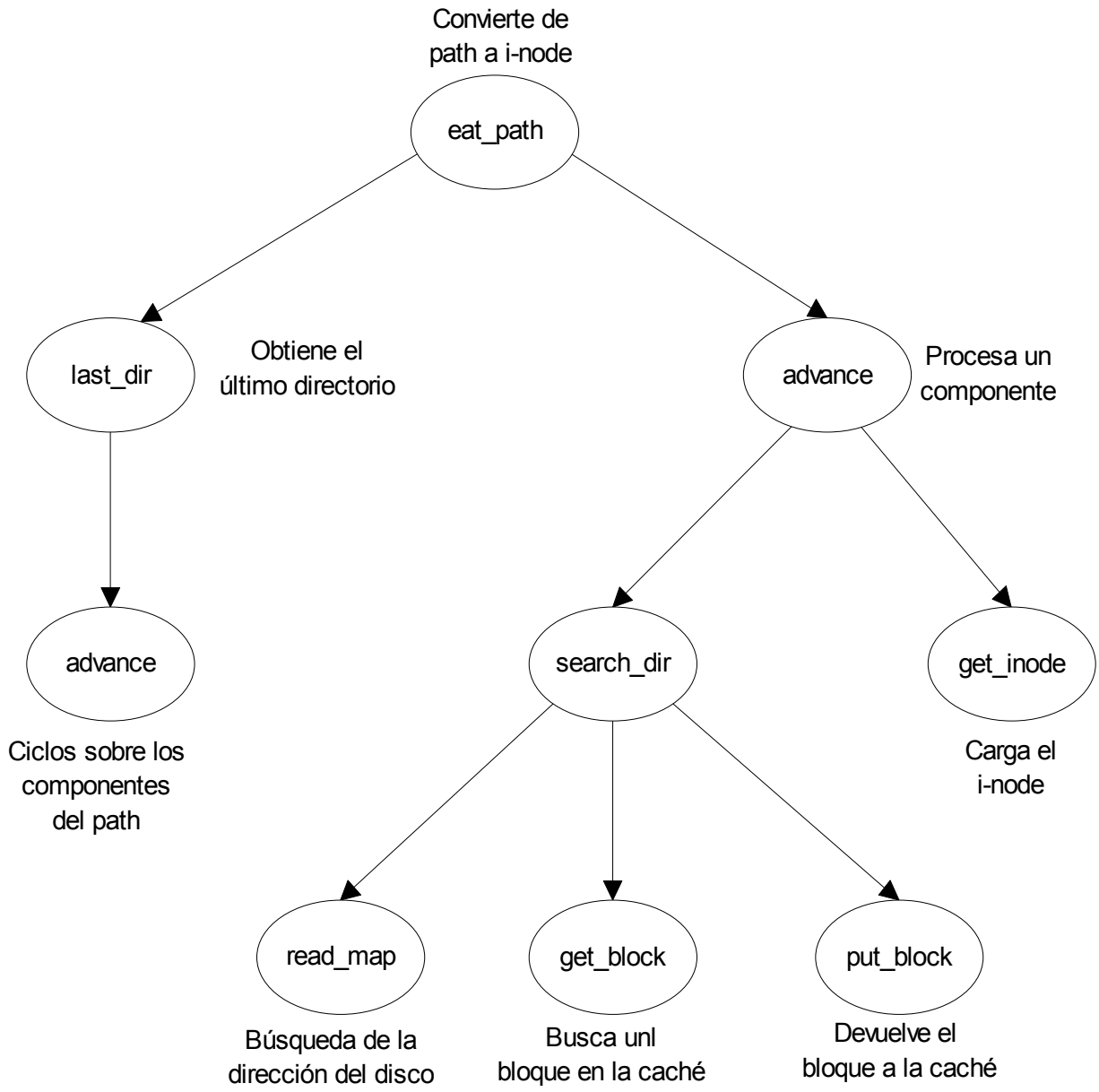


Figura 1.Algunos de los procedimientos usados en la búsqueda del path

3. Código fuente (path.c)

```
/* Este fichero contiene los procedimientos que buscan los  nombres del path en
el sistema de directorios y determina el  número del inode que está ligado a un
nombre de directorio dado.

   Los puntos de acceso al fichero son
    eat_path: La rutina principal del mecanismo de conversión de path a inode
    last_dir: Encuentra el directorio final en un path dado
    advance:  Analiza un componente de un nombre de path
    search_dir:      Recorre un directorio buscando una string y devuelve su
número de inode
*/

#include "fs.h"
#include <string.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

PUBLIC char dot1[2] = ".";/* used for search_dir to bypass the access */
PUBLIC char dot2[3] = "..";      /* permissions for . and ..                */

FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX])
);
```

3.1 EAT_PATH

- **Parámetros de entrada:**

-path: Puntero a la trayectoria a analizar sintácticamente.

- **Parámetros de salida:**

La salida de esta función será un puntero al i-node del fichero especificado en el path.

- **Funciones externas usadas:**

-put_inode(): Libera un i-node

- **Objetivo:**

Analiza gramaticalmente una ruta, colocando su i-node en la tabla de i-nodes y devolviendo un puntero al i-node.

- **Algoritmo:**

Llama a **lastdir**

Este le devuelve en ldip la dirección del i-node del penúltimo elemento de path y en

string el nombre del último elemento de path.)

Si le ha devuelto NIL_INODE (dirección nula de i-node) **entonces** retorna NIL_INODE.
Si string es nula **entonces** devuelve ldip.
Llama a **advance**

Calcula la dirección del i-node asociado al nombre en string.

Devuelve esta dirección.

- **Fuente:**

```
PUBLIC struct inode *eat_path(path)
char *path;               /* the path name to be parsed */
{
/*  Parse  the  path  'path'  and  put  its  inode  in  the  inode  table.  If  not
possible,
 * return NIL_INODE as function value and an error code in 'err_code'.
 */

  register struct inode *ldip, *rip;
  char string[NAME_MAX];  /* hold 1 path component name here */

  /* First open the path down to the final directory. */
  if ( (ldip = last_dir(path, string)) == NIL_INODE)
       return(NIL_INODE);  /* we couldn't open final directory */

  /* The path consisting only of "/" is a special case, check for it. */
  if (string[0] == '\0') return(ldip);

  /* Get final component of the path. */
  rip = advance(ldip, string);
  put_inode(ldip);
  return(rip);
}
```

3.2 LAST_DIR

- **Parámetros de entrada:**

-path: Puntero a la trayectoria a analizar.

- **Parámetros de salida:**

-string: En esta variable devuelve el último componente del path analizado. En general, será un nombre de fichero.

También devuelve un puntero al último directorio del path, o bien, si hubo error devuelve NIL_INODE.

- **Funciones externas usadas:**

-dup_inode(): Incrementa el contador de uso de un i-node.
-put_inode(): Libera un i-node

- **Objetivo:**

Esta función es llamada por eat_path. Dada una ruta, la analiza con objeto de obtener el i-node del último directorio que la compone. Devuelve la dirección de este i-node. En caso de que se produzca algún error devuelve NIL_INODE. Además, devuelve en la variable string el último componente del path analizado.

- **Algoritmo**

Mira si el path es absoluto o relativo e inicializa 'rip' en consecuencia.
Si el directorio se ha borrado o el path está vacio **entonces** devuelve ENOENT.
Mientras el path tenga componentes:
    Llama a **get_name** (lee el siguiente componente del path.)
    Si get_name devuelve un error **entonces**
        devuelve NIL_INODE y sale.
    Si get_name devuelve la string nula **entonces**
        Si rip es un direcctorio
            devuelve la última dirección calculada en rip y sale (salida normal).
        sino
            devuelve NIL_INODE
    Calcula en rip del nuevo elemento llamando a **advance**.
Fin mientras.

- **Fuente**

```
PUBLIC struct inode *last_dir(path, string)
char *path;               /* the path name to be parsed */
char string[NAME_MAX];          /* the final component is returned here */
{
/* Given a path, 'path', located in the fs address space, parse it as
 * far as the last directory, fetch the inode for the last directory into
 * the inode table, and return a pointer to the inode.  In
 * addition, return the final component of the path in 'string'.
 * If the last directory can't be opened, return NIL_INODE and
 * the reason for failure in 'err_code'.
 */

  register struct inode *rip;
  register char *new_name;
  register struct inode *new_ip;

  /* Is the path absolute or relative?  Initialize 'rip' accordingly. */
  rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);

  /* If dir has been removed or path is empty, return ENOENT. */
  if (rip->i_nlinks == 0 || *path == '\0') {
       err_code = ENOENT;
       return(NIL_INODE);
  }

  dup_inode(rip);         /* inode will be returned with put_inode */

  /* Scan the path component by component. */
```

```
  while (TRUE) {
       /* Extract one component. */
       if ( (new_name = get_name(path, string)) == (char*) 0) {
              put_inode(rip);     /* bad path in user space */
              return(NIL_INODE);
       }
       if (*new_name == '\0')
              if ( (rip->i_mode & I_TYPE) == I_DIRECTORY)
                     return(rip); /* normal exit */
              else {
                     /* last file of path prefix is not a directory */
                     put_inode(rip);
                     err_code = ENOTDIR;
                     return(NIL_INODE);
              }

       /* There is more path.  Keep parsing. */
       new_ip = advance(rip, string);
       put_inode(rip);            /* rip either obsolete or irrelevant */
       if (new_ip == NIL_INODE) return(NIL_INODE);

       /* The call to advance() succeeded.  Fetch next component. */
       path = new_name;
       rip = new_ip;
  }
}
```

3.3 GET_NAME

- **Parámetros de entrada:**

-old_name: Puntero a la trayectoria a analizar.

- **Parámetros de salida:**

-string: En esta variable devuelve la componente extraida de old_name.

También devuelve un puntero a la parte de old_name que todavía no ha sido analizada. get_name = old_name - string.

- **Funciones externas usadas:**

No utiliza funciones externas.

- **Objetivo:**

Dado un path, copia la primera componente a string. Además, devuelve un puntero a la parte del path que todavía no ha sido analizada.

- **Algoritmo:**

- Recorre el path con el puntero rnp desde su comienzo leyendo todos los '/' iniciales.

- Copia el primer elemento caracter a caracter a string.

- Para hacer /usr/ast/ equivalente a /usr/ast, quita los / finales. rnp apuntará al comienzo del path que queda sin analizar.

- **Si** rnp apunta a un dato incorrecto **entonces**
devuelve un error, sino devuelve rnp

- **Fuente:**

```
PRIVATE char *get_name(old_name, string)
char *old_name;                 /* path name to parse */
char string[NAME_MAX];          /* component extracted from 'old_name' */
{
/* Given a pointer to a path name in fs space, 'old_name', copy the next
 * component to 'string' and pad with zeros.  A pointer to that part of
 * the name as yet unparsed is returned.  Roughly speaking,
 * 'get_name' = 'old_name' - 'string'.
 *
 * This routine follows the standard convention that /usr/ast, /usr//ast,
 * //usr///ast and /usr/ast/ are all equivalent.
 */

  register int c;
  register char *np, *rnp;

  np = string;                  /* 'np' points to current position */
  rnp = old_name;         /* 'rnp' points to unparsed string */
  while ( (c = *rnp) == '/') rnp++;     /* skip leading slashes */

  /* Copy the unparsed path, 'old_name', to the array, 'string'. */
  while ( rnp < &old_name[PATH_MAX]  &&  c != '/'   &&  c != '\0') {
       if (np < &string[NAME_MAX]) *np++ = c;
       c = *++rnp;         /* advance to next character */
  }

  /* To make /usr/ast/ equivalent to /usr/ast, skip trailing slashes. */
  while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;

  if (np < &string[NAME_MAX]) *np = '\0';     /* Terminate string */

  if (rnp >= &old_name[PATH_MAX]) {
       err_code = ENAMETOOLONG;
       return((char *) 0);
  }
  return(rnp);
}
```

3.4 ADVANCE

- **Parámetros de entrada.**

dirp : Puntero al nodo-i del directorio en el que se va a buscar la componente del path data.
string : Componente del path que se va a buscar en el directorio dado por **dirp**.

- **Parámetros de salida.**

Devuelve por medio de return un puntero al nodo-i de la componente que se buscó en el directorio. Si hubo error devuelve NIL_INODE.

- **Funciones externas usadas.**

get_inode : Busca en la tabla de nodos el nodo-i dado. Si lo encuentra incrementa su contador de uso y devuelve su dirección. Si no lo encuentra, lo carga y devuelve igualmente su dirección.
put_inode : Libera un nodo-i. Decrementa el contador de uso y si es cero y ha sido modificado lo escribe en disco.

- **Objetivo**

Dado un directorio y una componente de un path, busca la componente en el directorio, obtiene su nodo-i, lo carga en la tabla de nodos y devuelve un puntero a él. Si no se puede hacer devuelve NIL_INODE.

- **Algoritmo**

Si 'string' está vacía **entonces**
devolver el mismo puntero al nodo-i que nos han pasado.
Si el nodo-i del directorio es NIL-INODE **entonces**
devolver NIL-INODE.
Llamar a search_dir para buscar la string en el directorio.
Si no está **entonces** devolver ERROR.
Si la string es .. y estamos en un nodo raiz **entonces**
devolver el mismo puntero al nodo-i que nos han pasado.
Si no se puede obtener la dirección del i-node **entonces**
devolver NIL-INODE.
Si los nodos-i del directorio y de string son nodos raíz y string[1]='.' **entonces**
Liberar el nodo-i de la raíz.
Reemplazarlo por el nodo-i que se ha montado.
Llamar a Advance
Si el nodo-i devuelto por Advance es NIL-INODE **entonces**
devolver NIL-INODE.
Si el nodo-i está montado **entonces**
cambiar al directorio raíz del fs montado buscando en la tabla de superbloque
Devolver el puntero al nodo-i.

- **Fuente**

```
PUBLIC struct inode *advance(dirp, string)
struct inode *dirp;       /* inode for directory to be searched */
char string[NAME_MAX];          /* component name to look for */
{
/* Given a directory and a component of a path, look up the component in
 * the directory, find the inode, open it, and return a pointer to its inode
 * slot.  If it can't be done, return NIL_INODE.
 */

  register struct inode *rip;
  struct inode *rip2;
  register struct super_block *sp;
  int r, inumb;
  dev_t mnt_dev;
  ino_t numb;

  /* If 'string' is empty, yield same inode straight away. */
  if (string[0] == '\0') return(get_inode(dirp->i_dev, (int) dirp->i_num));

  /* Check for NIL_INODE. */
  if (dirp == NIL_INODE) return(NIL_INODE);

  /* If 'string' is not present in the directory, signal error. */
  if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
       err_code = r;
       return(NIL_INODE);
  }

  /* Don't go beyond the current root directory, unless the string is dot2. */
  if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
             return(get_inode(dirp->i_dev, (int) dirp->i_num));

  /* The component has been found in the directory.  Get inode. */
  if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE)
       return(NIL_INODE);

  if (rip->i_num == ROOT_INODE)
       if (dirp->i_num == ROOT_INODE) {
           if (string[1] == '.') {
              for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
                     if (sp->s_dev == rip->i_dev) {
                            /* Release the root inode.  Replace by the
                             * inode mounted on.
                             */
                            put_inode(rip);
                            mnt_dev = sp->s_imount->i_dev;
                            inumb = (int) sp->s_imount->i_num;
                            rip2 = get_inode(mnt_dev, inumb);
                            rip = advance(rip2, string);
                            put_inode(rip2);
                            break;
                     }
              }
           }
       }
  if (rip == NIL_INODE) return(NIL_INODE);

  /* See if the inode is mounted on.  If so, switch to root directory of the
   * mounted file system.  The super_block provides the linkage between the
   * inode mounted on and the root directory of the mounted file system.
   */
  while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
       /* The inode is indeed mounted on. */
       for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
              if (sp->s_imount == rip) {
                     /* Release the inode mounted on.  Replace by the
                      * inode of the root inode of the mounted device.
                      */
```

```
                    put_inode(rip);
                    rip = get_inode(sp->s_dev, ROOT_INODE);
                    break;
                }
            }
        }
    return(rip);            /* return pointer to inode's component */
}
```

3.5 SEARCH_DIR

- **Parámetros de entrada**

ldir_ptr: Puntero al i-node del directorio en donde se buscará.
string : Componente que se desea buscar.
flag : Indicará la operación a realizar, los valores permitidos serán :
LOOK_UP busca la cadena.
ENTER introduce la cadena.
DELETE elimina la cadena.
IS_EMPTY comprueba si un directorio está vacio

- **Parámetros de salida**

numb : Puntero al número del i-node encontrado.
OK : Cuando la operación se ha realizado correctamente.
err_code : Caso de que la operación no se realice correctamente se cargará un codigo de error.

- **Funciones externas**

get_block : Busca un bloque de lectura o escritura en la lista de reserva de bloques. Cuando se hace una petición de bloque, se realiza una verificación de existencia del bloque en la cache. Si el bloque no está se trae, y se pasa la dirección del bloque asignado.
new_block : Adquiere un nuevo bloque y devuelve un puntero hacia este bloque.
read_map : Dado un i-node y una posición dentro del archivo, localiza el número del bloque en el que se halla dicha posición.
put_block : Devuelve un bloque a la lista LRU, bloque que previamente había sido pedido con un get block().
forbiden : Verifica si el acceso está permitido.
get_inode : Busca en la tabla de nodos el nodo-i dado. Si lo encuentra incrementa su contador de uso y devuelve su dirección. Si no lo encuentra, lo carga y devuelve igualmente su dirección.
put_inode : Libera un nodo-i. Decrementa el contador de uso y si es cero y ha sido modificado lo escribe en disco.

- **Objetivos**

Buscar en un directorio una string, y caso de que la encuentre retornar un puntero al número de i-node. Esta función será utilizada por otras funciones del sistema de archivos para realizar además otras tres operaciones: el borrado y la adición de un fichero, y la combrobación de si un directorio está vacio.

- **Algoritmo**

Si 'ldir-ptr' no es un puntero a un nodo-i de directorio **entonces**
devolver ERROR.
Comprueba los permisos según el tipo de operación.
Para cada bloque del directorio **hacer**
obtener el bloque
Para cada ranura dentro del bloque **hacer**
Si no hemos encontrado la string pero queda sitio en el bloque y es ENTER **entonces** break.
Si no ENTER **entonces**
Si es IS_EMPTY **entonces**
Si la ranura no contiene '.' o '..' **entonces** emparejamiento realizado.
sino
Si la ranura contiene a string **entonces**
emparejamiento realizado.
Si emparejamiento realizado **entonces**
Si IS_EMPTY **entonces**
devolver ENOTEMPTY
sino Si DELETE **entonces**
salvar el número de nodo-i para recuperar.
borrar la entrada.
sino
obtener el puntero al número del nodo-i de la string.
Si ENTER y la ranura está libre **entonces**
se puede realizar el ENTER más adelante. break.
Si no ENTER **entonces** devolver ERROR..
Si no se ha encontrado ninguna ranura libre **entonces**
intentar ensanchar el directorio
Si no se puede **entonces** devolver ERROR.
Relizar la entrada de string en la ranura libre.
Devolver OK.

- **Fuente**

```
PUBLIC int search_dir(ldir_ptr, string, numb, flag)
register struct inode *ldir_ptr; /* ptr to inode for dir to search */
char string[NAME_MAX];           /* component to search for */
ino_t *numb;              /* pointer to inode number */
      int flag;                  /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
{
/* This function searches the directory whose inode is pointed to by 'ldip':
 * if (flag == ENTER)  enter 'string' in the directory with inode # '*numb';
 * if (flag == DELETE) delete 'string' from the directory;
```

```
* if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
* if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
*
*    if 'string' is dot1 or dot2, no access permissions are checked.
*/

 register struct direct *dp;
 register struct buf *bp;
 int i, r, e_hit, t, match;
 mode_t bits;
 off_t pos;
 unsigned new_slots, old_slots;
 block_t b;
 struct super_block *sp;
 int extended = 0;

 /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
 if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);

 r = OK;

 if (flag != IS_EMPTY) {
      bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);

      if (string == dot1 || string == dot2) {
            if (flag != LOOK_UP) r = read_only(ldir_ptr);
                              /* only a writable device is required. */
        }
      else r = forbidden(ldir_ptr, bits); /* check access permissions */
 }
 if (r != OK) return(r);

 /* Step through the directory one block at a time. */
 old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
 new_slots = 0;
 e_hit = FALSE;
 match = 0;              /* set when a string match occurs */

 for (pos = 0; pos < ldir_ptr->i_size; pos += BLOCK_SIZE) {
      b = read_map(ldir_ptr, pos);      /* get block number */

      /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
      bp = get_block(ldir_ptr->i_dev, b, NORMAL);   /* get a dir block */

      /* Search a directory block. */
      for (dp = &bp->b_dir[0]; dp < &bp->b_dir[NR_DIR_ENTRIES]; dp++) {
            if (++new_slots > old_slots) { /* not found, but room left */
                  if (flag == ENTER) e_hit = TRUE;
                  break;
            }

            /* Match occurs if string found. */
            if (flag != ENTER && dp->d_ino != 0) {
                  if (flag == IS_EMPTY) {
                        /* If this test succeeds, dir is not empty. */
                        if (strcmp(dp->d_name, "." ) != 0 &&
                            strcmp(dp->d_name, "..") != 0) match = 1;
                  } else {
                        if (strncmp(dp->d_name, string, NAME_MAX) == 0)
                              match = 1;
                  }
            }

            if (match) {
                  /* LOOK_UP or DELETE found what it wanted. */
                  r = OK;
                  if (flag == IS_EMPTY) r = ENOTEMPTY;
                  else if (flag == DELETE) {
                        /* Save d_ino for recovery. */
                        t = NAME_MAX - sizeof(ino_t);
                        *((ino_t *) &dp->d_name[t]) = dp->d_ino;
                        dp->d_ino = 0;      /* erase entry */
```

```
                        bp->b_dirt = DIRTY;
                        ldir_ptr->i_update |= CTIME | MTIME;
                        ldir_ptr->i_dirt = DIRTY;
                } else {
                        sp = ldir_ptr->i_sp;      /* 'flag' is LOOK_UP */
                        *numb = conv2(sp->s_native, (int) dp->d_ino);
                }
                put_block(bp, DIRECTORY_BLOCK);
                return(r);
        }


        /* Check for free slot for the benefit of ENTER. */
        if (flag == ENTER && dp->d_ino == 0) {
                e_hit = TRUE;/* we found a free slot */
                break;
        }
    }

    /* The whole block has been searched or ENTER has a free slot. */
    if (e_hit) break;   /* e_hit set if ENTER can be performed now */
    put_block(bp, DIRECTORY_BLOCK);  /* otherwise, continue searching dir */
}

/* The whole directory has now been searched. */
if (flag != ENTER) return(flag == IS_EMPTY ? OK : ENOENT);

/* This call is for ENTER.  If no free slot has been found so far, try to
 * extend directory.
 */
if (e_hit == FALSE) { /* directory is full and no room left in last block */
    new_slots++;        /* increase directory size by 1 entry */
    if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
    if ( (bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
          return(err_code);
    dp = &bp->b_dir[0];
    extended = 1;
}

/* 'bp' now points to a directory block with space. 'dp' points to slot. */
(void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
sp = ldir_ptr->i_sp;
dp->d_ino = conv2(sp->s_native, (int) *numb);
bp->b_dirt = DIRTY;
put_block(bp, DIRECTORY_BLOCK);
ldir_ptr->i_update |= CTIME | MTIME; /* mark mtime for update later */
ldir_ptr->i_dirt = DIRTY;
if (new_slots > old_slots) {
    ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
    /* Send the change to disk if the directory is extended. */
    if (extended) rw_inode(ldir_ptr, WRITING);
}
return(OK);
```

4. Cuestiones

1. ¿Cuál es es objetivo general del path.c?
2. Explica los procedimientos principales por los que se pasa cuando se realiza una llamada a eat_path.

5.Apéndice

Traza del ejemplo que se explicó en el punto 1.Introducción.

```
inode=eat_path(path='/usr/ast/mbox')
    ldip=last_dir(path='/usr/ast/mbox')
        rip=fp_rootdir
        while(TRUE)
            new_name=get_name(old_name='/usr/ast/mbox',strin
            g)
            ← string='usr'
            ← new_name='/ast/mbox'
            ¿new_name==''? → NO
            new_ip=advance(dirp=fp_rootdir,string='usr')
                r=search_dir(dirp=fp_rootdir,string='usr',
                numb,LOOK_UP)
                    para cada entrada de cada bloque
                        si d_name==string
                            numb=conv2()
                ← numb=6
                ← r=OK
                rip=get_inode(deb,numb)=^6
            ← new_ip=^6
            path=new_name='/ast/mbox'
            rip=new_ip=^6
            new_name=get_name(old_name='/ast/mbox',string)
            ← string='ast'
            ← new_name='/mbox'
            ¿new_name==''? → NO
            new_ip=advance(dirp=^6,string='ast')
                r=search_dir(dirp=^6,string='ast',numb,LOO
                K_UP)
                    para cada entrada de cada bloque
                        si d_name==string
                            numb=conv2()
                ← numb=26
                ← r=OK
                rip=get_inode(deb,numb)=^26
            ← new_ip=^26
            path=new_name='/mbox'
            rip=new_ip=^26
            new_name=get_name(old_name='/mbox',string)
            ← string='mbox'
            ← new_name=''
            ¿new_name==''? → SI
    ← ldip=^26
    rip=advance(dirp=^26,string='mbox')
        r=search_dir(dirp=^26,string='usr',numb,LOOK_UP)
            para cada entrada de cada bloque
                si d_name==string
                    numb=conv2()
        ← numb=60
        ← r=OK
        rip=get_inode(deb,numb)=^60
    ← rip=^60
```

← `rip=^60`