# Cambio de directorio

# STADIR

Nicolás Hdez Guerra de Aguilar
Guayasen González Santiago

# STADIR.C

# INTRODUCCIÓN

El archivo STADIR.C contiene el código de cuatro llamadas al sistema:

**-CHDIR**

**-CHROOT**

**-STAT**

**-FSTAT**

Cada proceso tiene asignados un directorio de trabajo actual (nombre de ruta relativo) y un directorio raiz (nombre de ruta absoluto). Por lo tanto, existen dos punteros que hace referencia al nodo-i de dichos directorios. El motivo por el cual existe un puntero al directorio raiz, se debe a que algunos programas necesitan acceder a un archivo específico sin considerar el directorio actual de trabajo, además, el directorio raíz no siempre se encuentra en /. El nombre de ruta absoluta siempre funciona, sea cual sea el directorio de trabajo. Evidentemente, si el proceso necesita una gran cantidad de archivos situados en otro directorio, en lugar de emplear nombres de rutas absolutas, una estrategia alternativa es que emita una llamada al sistema para cambiar el directorio de trabajo y así poder utilizar nombres de rutas relativas.

El cambio de un directorio de trabajo (o directorio raíz) a otro es simplemente cuestión de cambiar éstos punteros dentro de la tabla de procesos del solicitante. Estos cambios los efectúan DO_CHDIR y DO_CHROOT . Ambos realizan la verificación necesaria y después llaman a CHANGE para abrir el nuevo directorio a fin de sustituir el anterior.

Para cada archivo, Minix lleva el control del modo del archivo (archivo regular, especial, directorio, etc...), tamaño, tiempo de la última modificación y otra información. Los procesos pueden solicitar ver esta información mediante las llamadas al sistema STAT y FSTAT. Estas difieren sólo en que la primera especifica el archivo por nombre, mientras que la segunda toma un descriptor del archivo.

Ambas llamadas ofrecen en el segundo parámetro un puntero a una estructura donde se colocará la información. Esta estructura es demasiado grande como para poderla pasar como parámetro asi que se almacena en el espacio de usuario.

Esta estructura es la siguiente:

```
struct stat {
      short st_dev;              /* dispositivo que contiene el FS */
      unsigned short st_ino;     /* número del nodo-i en ese dispositivo*/
      unsigned short st_mode;    /* palabra de modo (tipo y acceso) */
      short st_nlink;            /* número de enlaces */
      short st_uid;              /* Id. del usuario propietario*/
      short st_gid;              /* Id. del grupo al que pertenece*/
      short st_rdev;             /* dispositivo mayor/menor de
                                    archivos especiales */
      long st_size;              /* tamaño del archivo */
      long st_atime;             /* hora de la última modificación */
      long st_mtime;             /* igual a st_atime */
      long st_ctime;             /* igual a st_atime */
}
```

En Minix los tres tiempos son idénticos y se ofrece para que haya compatibilidad con Unix, donde si son diferentes.

# PROCEDIMIENTOS EXTERNOS UTILIZADOS:

***dup_inode(rip)***: [fs/inode.c]

Utilizado para avisar que alguien más está utilizando ese mismo nodo-i. Incrementa el contador del nodo-i que indica cuantas veces se encuentra abierto el archivo al que pertenece. Esto evita traer a memoria varios nodos-i del mismo archivo.

***eat_path(user_path)***: [fs/path.c]

Analiza sintácticamente *user_path*, carga su nodo-i en memoria y retorna un apuntador al i-nodo; de no ser posible, devuelve *NIL_INODE* como valor de la función y un código de error en *err_code*.

***fetch_name(name_ptr, len, flag)***: [fs/utility.c]

Toma la trayectoria especificada, y la coloca en la variable global *user_path*. Si *flag = M3*, y la longitud (*len*) es menor o igual a 14 bytes, la trayectoria está presenta en el mensaje; si no la cadena no está contenida en el mensaje y se coge del espacio de usuario.

***forbidden(rip, X_BIT)***: [fs/protect.c]

Dado un apuntador a un nodo-i, y el acceso deseado, determina si está permitido el acceso. Si se permite el acceso, se devuelve *OK*, si no se devuelve *EACCES*. En nuestro caso, el acceso deseado es de ejecución que viene dado por el flag *X_BIT*.

***get_filp(fd)***: [fs/filedes.c]

Comprueba si *fd* es un descriptor de archivo valido, si es así devuelve un puntero a fd, si no devuelve *NIL_FILP*.

***put_inode(rip)***: [fs/inode.c]

Libera el nodo-i *rip*. Si nadie más lo esta usando, lo escribe en el disco. Si no tiene enlaces, lo marca como disponible.

***Sys_copy(FS_PROC_NR, D, &statbuf, who D, user_addr, sizeof(statbuf));***

Utilizado para copiar información de un buffer, al espacio de usuario, debido a que es demasiado grande para caber en un mensaje.

## DO_CHDIR

Esta función presenta lineas de código para las llamadas efectuadas por el Manejador de Memoria, con el objeto de cambiar a un directorio de usuario a fin de manejar llamadas EXEC. Cuando un usuario intenta ejecutar un archivo, por ejemplo, **a.out** en su directorio de trabajo, resulta más sencillo para el manejador de la memoria cambiar a ese directorio que intentar imaginar en donde se encuentra. Asi que lo que hace es apuntar el puntero del directorio actual de trabajo *fp* al mismo lugar donde apunta el puntero de directorio de trabajo del proceso *rfp*:

fp->fp_workdir = rfp->fp_workdir

Y lo mismo con el directorio raiz:

fp->fp_rootdir = rfp->fp_rootdir

**ALGORITMO:**

**SI** (proceso solicitante es el MM) **ENTONCES**

- Actualizamos los campos fp_rootdir y fp_workdir del nodo-i cargado.
- Liberamos los nodos-i que teníamos (**PUT_INODE**).
- Actualizamos los nuevos nodos-i (**DUP_INODE**).
-Cambiamos el identificador de usuario efectivo y real al del nuevo nodo-i.
-Retornamos (**OK**).

**FINSI**

- Llamamos a la función CHANGE para realizar el cambio del campo fp_workdir.
- Retornar el resultado de la llamada.

## DO_CHROOT

Esta función cambia el directorio raíz. Esta llamada al sistema únicamente puede ser realizada por el Superusuario.

**ALGORITMO:**

**SI** (no superusuario) **ENTONCES**

retornar (error)

**FINSI**

- Llamamos a la función CHANGE para realizar el cambio del campo fp_rootdir.

## CHANGE

Para realizar los cambios de directorios los procedimientos **do_chdir** y **do_chroot** realizan la verificación necesaria siendo realmente la función **change** la que abre el nuevo directorio a fin de sustituir el anterior.

**ALGORITMO:**

-Dado el nombre del archivo *name_ptr*, se obtiene la ruta mediante la función *fetch_name* y se almacena en la variable global *user_path*.

- Analiza la trayectoria (*eat_path()*), y recibimos un puntero al nodo-i que corresponde a la ruta especificada.

- Comprobamos que sea un directorio, y que esté permitido su acceso (*forbidden()*).

- Finalmente, si no hubo errores, liberamos el viejo nodo-i y tomamos el nuevo.

## DO_STAT

Como se mencionó en la introducción, para cada archivo Minix se lleva el control de modo del archivo (archivo regular, especial, directorio, etc...), tamaño, tiempo de la última modificación y otra información. Los procesos pueden solicitar ver esta información mediante las llamadas al sistema STAT y FSTAT.

El procedimiento STAT especifica el archivo por nombre. Realiza la llamada al sistema `STAT(name, buf)`, donde *name* es la ruta y el nombre del archivo del que se quiere obtener la información. Buf es un buffer donde almacenar los resultados de la llamada, es decir, la estructura stat.

**ALGORITMO:**

-Tomamos la trayectoria especificada (*fetch_name()*), y la colocamos en *user_path*.

-Obtenemos un puntero al nodo-i (con *eat_path*), que corresponde con la trayectoria especificada en user_path.

-Se llama a la función *stat_inode*, para que realice el trabajo.

-Liberamos el nodo-i.

## DO_FSTAT

La otra llamada al sistema que proporciona la información a los procesos sobre el control del modo de cada archivo es la FSTAT.

El procedimiento FSTAT especifica el archivo por medio del descriptor del archivo.

**ALGORITMO:**

-Chequea si el descriptor del archivo es válido (*get_filp(fd)*), en cuyo caso obtiene un puntero a éste.
-Llama a la función `stat_inode` para que realice el trabajo. Sólo se le pasa el apuntador del nodo-i obtenido de la función anterior.

## STAT_INODE

Extrae información del nodo-i y la copia en un buffer. Este buffer se copia explícitamente en el espacio del usuario ya que es demasiado grande para caber en un mensaje.

**ALGORITMO:**

-Actualiza, si es necesario, los campos atime, ctime y mtime del nodo-i.
-Rellena la estructura del buffer de estado
-Copia la estructura en el espacio de usuario.

```
/* This file contains the code for performing four system calls
relating to
 * status and directories.
 *
 * The entry points into this file are
 *   do_chdir:    perform the CHDIR system call
 *   do_chroot:   perform the CHROOT system call
 *   do_stat:     perform the STAT system call
 *   do_fstat:    perform the FSTAT system call
 */

#include "fs.h"
#include <sys/stat.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

FORWARD _PROTOTYPE( int change, (struct inode **iip, char *name_ptr,
int len));
FORWARD _PROTOTYPE( int stat_inode, (struct inode *rip, struct filp
*fil_ptr,
                  char *user_addr)                    );

/*===========================================================================
=======*
 *                      do_chdir                             *

*===========================================================================
======*/
PUBLIC int do_chdir()
{
/* Change directory.  This function is  also called by MM to simulate
a chdir
 * in order to do EXEC, etc.  It also changes the root directory, the
uids and
 * gids, and the umask.
 */

  int r;
  register struct fproc *rfp;

  if (who == MM_PROC_NR) {
      rfp = &fproc[slot1];
      put_inode(fp->fp_rootdir);
      dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
      put_inode(fp->fp_workdir);
      dup_inode(fp->fp_workdir = rfp->fp_workdir);

      /* MM uses access() to check permissions.  To make this work,
pretend
       * that the user's real ids are the same as the user's effective
ids.
       * FS calls other than access() do not use the real ids, so are
not
       * affected.
       */
      fp->fp_realuid =
      fp->fp_effuid = rfp->fp_effuid;
      fp->fp_realgid =
      fp->fp_effgid = rfp->fp_effgid;
      fp->fp_umask = rfp->fp_umask;
      return(OK);
  }

  /* Perform the chdir(name) system call. */
```

```
  r = change(&fp->fp_workdir, name, name_length);
  return(r);
}


/*===========================================================================
=======*
 *                      do_chroot                            *

*===========================================================================
======*/
PUBLIC int do_chroot()
{
/* Perform the chroot(name) system call. */

  register int r;

  if (!super_user) return(EPERM);   /* only su may chroot() */
  r = change(&fp->fp_rootdir, name, name_length);
  return(r);
}


/*===========================================================================
=======*
 *                      change                                  *

*===========================================================================
======*/
PRIVATE int change(iip, name_ptr, len)
struct inode **iip;           /* pointer to the inode pointer for the
dir */
char *name_ptr;               /* pointer to the directory name to
change to */
int len;                /* length of the directory name string */
{
/* Do the actual work for chdir() and chroot(). */

  struct inode *rip;
  register int r;

  /* Try to open the new directory. */
  if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* It must be a directory and also be searchable. */
  if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
      r = ENOTDIR;
  else
      r = forbidden(rip, X_BIT);    /* check if dir is searchable */

  /* If error, return inode. */
  if (r != OK) {
      put_inode(rip);
      return(r);
  }

  /* Everything is OK.  Make the change. */
  put_inode(*iip);            /* release the old directory */
  *iip = rip;                 /* acquire the new one */
  return(OK);
}


/*===========================================================================
=======*
```

```
 *                      do_stat                                         *

*==========================================================================
======*/
PUBLIC int do_stat()
{
/* Perform the stat(name, buf) system call. */

  register struct inode *rip;
  register int r;

  /* Both stat() and fstat() use the same routine to do the real work.
That
   * routine expects an inode, so acquire it temporarily.
   */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
  r = stat_inode(rip, NIL_FILP, name2);  /* actually do the work.*/
  put_inode(rip);       /* release the inode */
  return(r);
}


/*==========================================================================
=======*
 *                      do_fstat                                *

*==========================================================================
======*/
PUBLIC int do_fstat()
{
/* Perform the fstat(fd, buf) system call. */

  register struct filp *rfilp;

  /* Is the file descriptor valid? */
  if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);

  return(stat_inode(rfilp->filp_ino, rfilp, buffer));
}


/*==========================================================================
=======*
 *                      stat_inode                              *

*==========================================================================
======*/
PRIVATE int stat_inode(rip, fil_ptr, user_addr)
register struct inode *rip;   /* pointer to inode to stat */
struct filp *fil_ptr;         /* filp pointer, supplied by 'fstat' */
char *user_addr;        /* user space address where stat buf goes */
{
/* Common code for stat and fstat system calls. */

  struct stat statbuf;
  mode_t mo;
  int r, s;

  /* Update the atime, ctime, and mtime fields in the inode, if need
be. */
  if (rip->i_update) update_times(rip);

  /* Fill in the statbuf struct. */
  mo = rip->i_mode & I_TYPE;
```

```
  s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL);     /* true iff
special */
  statbuf.st_dev = rip->i_dev;
  statbuf.st_ino = rip->i_num;
  statbuf.st_mode = rip->i_mode;
  statbuf.st_nlink = rip->i_nlinks & BYTE;
  statbuf.st_uid = rip->i_uid;
  statbuf.st_gid = rip->i_gid & BYTE;
  statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
  statbuf.st_size = rip->i_size;

  if (rip->i_pipe == I_PIPE) {
      statbuf.st_mode &= ~I_REGULAR;     /* wipe out I_REGULAR bit for
pipes */
      if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
            statbuf.st_size -= fil_ptr->filp_pos;
  }

  statbuf.st_atime = rip->i_atime;
  statbuf.st_mtime = rip->i_mtime;
  statbuf.st_ctime = rip->i_ctime;

  /* Copy the struct to user space. */
  r = sys_copy(FS_PROC_NR, D, (phys_bytes) &statbuf,
           who, D, (phys_bytes) user_addr, (phys_bytes)
sizeof(statbuf));
  return(r);
}
```