utility
put
trace

Fco Javier Arbelo Robaina
Alexis Quintero Jiménez

Índice

1 Utility.c

Utility.c es un fichero que contiene diferentes rutinas utilizadas por el MM, en concreto se trata de cuatro:

- allowed
- no_sys
- panic
- tell_fs.

Las rutinas de este fichero no guardan ninguna iterelación, tan solo son rutinas necesitadas por el MM, y que ubica junto dentro de este archivo.

1.1 Cabecera del fichero

```
/* Este archivo contiene algunas rutinas de utilidad para el manejador de memoria (MM).
 * Los puntos de entrada son:
 *   allowed:     comprueba si se permite el acceso.
 *   no_sys:      esta rutina se llama para números inválidos de llamadas al sistema
 *   panic:       el manejador de memoria ha producido algún error fatal y no puede
 *                continuar.
 *   tell_fs:     interfaz con el sistema de ficheros (FS).
 */


#include "mm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <fcntl.h>
#include <signal.h>             /* necesaria tan sólo porque mproc.h la necesita */
#include "mproc.h"
```

A continuación mostraremos la finalidad y funcionamientocada una de estas funciones.

1.2 allowed()

Esta función verifica la accesibilidad de los ficheros. Para ello en primer luga comprueba el tipo de fichero y si se permite el acceso al mismo solisitado. En caso tratarse de un acceso prohibido, éste es denegado al usuario y la

función devuelve los valores EACCES o ENOENT. Si el acceso es legal la función devuelve al usuario el descriptor de fichero.

Allowed utiliza la función, fstat(). Se trata de una llamada al sistema operativo que pregunta por el tipo de fichero, tamaño, tiempo de la última modificación y otras estadisticas. Es caso de que esta esta función retorne un valor negativo allowed interpreta que se ha producido un grave problema e invoca a la función panic().

1.2.1 Código

```
/*===========================================================
                                        allowed
 ===========================================================*/
PUBLIC int allowed(name_buf, s_buf, mask)
char *name_buf;                         /* puntero al nombre del archivo que va a ser
'EXECutado' */
struct stat *s_buf;             /* buffer para hacer y retornar la estructura stat */
int mask;                       /* R_BIT, W_BIT, o X_BIT */
{

/* Comprobación para ver si un archivo puede ser accedido. Devuelve EACCES o
 * ENOENT si el acceso estó prohibido. Si el acceso es legal abre el archivo y
 * devuelve un descriptor de archivo.
 */

  int fd;
  int save_errno;

  /* Use the fact that mask for access() is the same as the permissions mask.
   * E.g., X_BIT in <minix/const.h> is the same as X_OK in <unistd.h> and
   * S_IXOTH in <sys/stat.h>.  tell_fs(DO_CHDIR, ...) has set MM's real ids
   * to the user's effective ids, so access() works right for setuid programs.
   */

  /* Utiliza el hecho de que la máscara para access() es la misma que la máscara de
   * permisos. Ej: X_BIT en <minix/const.h> es el mimo que X_OK en <unistd.h> y
   * S_IXOTH in <sys/stat.h>. tell_fs(DO_CHDIR, ...) ha seleccionado la identificación
   * real del manejador de memoria a la identificación efectiva del usuario, así que
   * access() trabaja correctamente para los programas setuid.
   */


  if (access(name_buf, mask) < 0) return(-errno);
```

```
  /* El archivo es accesible pero puede no ser legible. Hacerlo legible. */
  tell_fs(SETUID, MM_PROC_NR, (int) SUPER_USER, (int) SUPER_USER);


  /* Abrir el archivo y realizar fstat sobre él. Restaurar los identificadores
   * rápidamente para manejar los errores.
   */
  fd = open(name_buf, O_RDONLY);
  save_errno = errno;       /* open puede fallar, ej: de ENFILE */
  tell_fs(SETUID, MM_PROC_NR, (int) mp->mp_effuid, (int) mp->mp_effuid);
  if (fd < 0) return(-save_errno);
  if (fstat(fd, s_buf) < 0) panic("allowed: fstat failed", NO_NUM);

  /* Sólo los ficheros 'regulares' pueden ser ejecutados. */
  if (mask == X_BIT && (s_buf->st_mode & I_TYPE) != I_REGULAR) {
        close(fd);
        return(EACCES);
  }
  return(fd);
}
```

1.3 no_sys()

no_sys es invocada cuando un usuario invoca al administrador de la memoria con un numero de llamada al sistema no válido o interno al propio MM.

1.3.1 Código

```
/*===========================================================
                                        no_sys
 ===========================================================*/
PUBLIC int no_sys()
{
/* Se ha empleado un número de llamada al sistema no implementado en el MM*/
  return(EINVAL);
}
```

1.4 panic()

La función panic() es invocada sólo en caso de fallo del sistema, es decir, cuando es detectada una inconsistencia interna, como por ejemplo un error de programación o un valor ilegal en una constante definida. Panic informa del error a la tarea del sistema, la cual parará minix. Lo primero que realizará esta función es la presentación por pantalla del error que ha provocado el "pánico".

1.4.1 Código

```
/*===========================================================
                                panic
 ===========================================================*/
PUBLIC void panic(format, num)
char *format;                   /*cadena de formato*/
int num;                        /*número que va con la cadena de formato*/
{

/* Algo desagradable ha ocurrido. Los 'Panics' ocurren cuando se detecta una
 * inconsistencia interna, ej: un error de programación o un valor ilegal de una
 * constante definida.
 */


  printf("Panic del manejador de memoria: %s ", format);
  if (num != NO_NUM) printf("%d",num);
  printf("\n");
  tell_fs(SYNC, 0, 0, 0);       /* vuelca la caché en el disco */
  sys_abort(RBT_PANIC);
}
```

1.5 tell_fs()

Se trata de una rutina utilizada por el administrador de memoria MM para informar al sistema de ficheros FS de los siguientes eventos:

- CHDIR
- EXEC
- EXIT
- FORK
- SETGID
- SETSID
- SETUID
- SYNC
- UNPAUSE

1.5.1 Código

```
/*===========================================================
                                tell_fs
 ============================================================*/
PUBLIC void tell_fs(what, p1, p2, p3)
int what, p1, p2, p3;
{
/* Esta rutina sólo es empleada por el manejador de memoria para informar al sistema
 * de ficheros de ciertos eventos:
 *      tell_fs(CHDIR, slot, dir, 0)
 *      tell_fs(EXEC, proc, 0, 0)
 *      tell_fs(EXIT, proc, 0, 0)
 *      tell_fs(FORK, parent, child, pid)
 *      tell_fs(SETGID, proc, realgid, effgid)
 *      tell_fs(SETSID, proc, 0, 0)
 *      tell_fs(SETUID, proc, realuid, effuid)
 *      tell_fs(SYNC, 0, 0, 0)
 *      tell_fs(UNPAUSE, proc, signr, 0)
 */

  message m;
  m.m1_i1 = p1;
  m.m1_i2 = p2;
  m.m1_i3 = p3;
  _taskcall(FS_PROC_NR, what, &m);
}
```

2 Putk.c

Es un fichero que contiene dos rutinas utilizadas por el MM para la impresión ocacional de algunos mensajes. Se trata de las rutinas putk() y flush() que comentaremos a continuación. Estas rutinas son utilizadas por el procedimiento de biblioteca printk(). Es así que cuando el MM pretende mostrar algo por pantalla llama a la función printk(). La principal diferencia de esta función con printf(), es que esta última es una función utilizada desde procesos de usuario, la cual manda una petición al sistema operativo y este a su vez envía una solicitud de escritura por pantalla a la tarea de la terminal. En cambio printk() al ser invocado por el sistema lo requiere que su solicitud pase estos trámites (enviar la solicitud de impresión al FS), y en su lugar llama directamente a la tarea de la terminal (acción ilegal desde el contexto de usuario).

2.1 Cabecera del fichero:

```
/* El manejador de memoria (MM) debe imprimir algún mensaje de manera ocasional.
 * Utiliza la rutina estándar de librería printk(). (El nombre "printf" es en
 * realidad una macro definida como "printk"). La impresión del mensaje se realiza
 * mediante una llamada a la tarea de terminal (TTY) directamente, sin pasar a
 * través del sistema de ficheros (FS).
*/
#include "mm.h"
#include <minix/com.h>

#define BUF_SIZE        100  /* tamaño del buffer de impresión */

PRIVATE int buf_count;                  /* número de caracteres en el buffer */
PRIVATE char print_buf[BUF_SIZE];       /* la salida se mete en este buffer */
PRIVATE message putch_msg;              /* se utiliza para enviar mensajes a la tarea de terminal
                        (TTY task) */

_PROTOTYPE( FORWARD void flush, (void) );
```

2.2 putk()

Esta función acumula caracteres en un buffer de caracteres (propio para esta tarea del MM) hasta que se llena o recibe un 0. En esto ultimos casos llama a la función flush(), la cual se encarga de llamar a la tarea de la terminal para imprimir el contenido del buffer por pantalla.

2.2.1 Código

```
/*===========================================================
                                putk
 ===========================================================*/
PUBLIC void putk(c)
int c;
{
/* Acumular otro caracter.  Si 0 o buffer lleno, imprimirlo. */

  if (c == 0 || buf_count == BUF_SIZE) flush();
  if (c == '\n') putk('\r');
  if (c != 0) print_buf[buf_count++] = c;
}
```

2.3 flush()

flush() es una función, como ya hemos comentado con anterioridad, con la función de llamar a la tarea del sistema para imprimir el contenido del buffer de impresión del MM, y que ha ido llenando putk() a petición de printk().

2.3.1 Código

```
/*===========================================================
                                flush
 ===========================================================*/
PRIVATE void flush()
{
/* Realizar un volcado del buffer de impresión llamando a la tarea de terminal */
  if (buf_count == 0) return;
  putch_msg.m_type = DEV_WRITE;
  putch_msg.PROC_NR  = 0;
  putch_msg.TTY_LINE = 0;
  putch_msg.ADDRESS  = print_buf;
  putch_msg.COUNT = buf_count;
  sendrec(TTY, &putch_msg);
  buf_count = 0;
}
```

3 GetSet.c

Este archivo tan solo contiene una función, do_getset(), cuya misión es gestionar las llamadas al sistema que necesitan del MM para leer y escribir los distintos tipos de identificadores (UIDS, GIDS, PIDS).

Identificadores:

de proceso:	PID --> GETPID
de grupo:	GID --> GETGID y SETGID
de usuario:	UID --> GETUID y SETUID

Estas llamadas se encargan de obtener los PID, GID y UID. Por supuesto, para el identificador de proceso PID, no existe la posibilidad de cambiar su valor, ya que este identificador una vez establecido por el sistema no debe ser modificado. En cambio los identificadores de grupo y usuario si pueden ser modificados por el superusuario por lo que getset() dispone de dicha posibilidad mediante las llamada SETGID y SETUID.

3.1 Código

```
/* Este archivo maneja las cuatro llamadas al sistema que leen y actualizan identificadores
de usuario (uids) y de grupo (gids). Además, también maneja las funciones getpid(), setsid()
y getpgrp(). El código para cada función es tan pequeño que es innecesario realizar una
función separada para cada una, por lo cual han sido agrupadas en unsólo archivo.*/

#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/*===========================================================
*                                do_getset
*
*===========================================================*/
PUBLIC int do_getset()
{
/* Maneja GETUID, GETGID, GETPID, GETPGRP, SETUID, SETGID, SETSID. Los
cuatro GETs y SETSID devuelven sus resultados en 'r'. GETUID, GETGID, y GETPID
también devuelven resultados secundarios (la identificación efectiva(ID), o el identificador
del proceso padre) en 'result2', la cual es devuelta al usuario.*/
```

```
register struct mproc *rmp = mp;
register int r;

switch(mm_call) {
	case GETUID:
		r = rmp->mp_realuid;
		result2 = rmp->mp_effuid;
		break;

	case GETGID:
		r = rmp->mp_realgid;
		result2 = rmp->mp_effgid;
		break;

	case GETPID:
		r = mproc[who].mp_pid;
		result2 = mproc[rmp->mp_parent].mp_pid;
		break;

	case SETUID:
		if (rmp->mp_realuid != usr_id && rmp->mp_effuid!= SUPER_USER)
			return(EPERM);
		rmp->mp_realuid = usr_id;
		rmp->mp_effuid = usr_id;
		tell_fs(SETUID, who, usr_id, usr_id);
		r = OK;
		break;

	case SETGID:
		if (rmp->mp_realgid != grpid && rmp->mp_effuid != SUPER_USER)
			return(EPERM);
		rmp->mp_realgid = grpid;
		rmp->mp_effgid = grpid;
		tell_fs(SETGID, who, grpid, grpid);
		r = OK;
		break;

	case SETSID:
		if (rmp->mp_procgrp == rmp->mp_pid) return(EPERM);
		rmp->mp_procgrp = rmp->mp_pid;
		tell_fs(SETSID, who, 0, 0);
		/*FALL THROUGH*/

	case GETPGRP:
		r = rmp->mp_procgrp;
		break;
```

```
        default:
                r = EINVAL;
                break;
  }
  return(r);
}
```

4 Trace.c

Las funciones incluidas en este archivo permiten realizar la depuración de los programas, la cual se realiza mediante la llamada al sistema **ptrace**. Con ellas, se puede *congelar* la ejecución de un programa y continuar con la misma, así como obtener información sobre el estado del mismo.

El módulo consta de tres funciones (do_trace, findproc y stop_proc), las cuales describiremos más adelante.

4.1 Los comandos de debug

En este archivo están las funciones relacionadas con poner un proceso en modo traza. Comentaremos a continuación los diferentes comandos de debug que existen en relación con la traza de un proceso:

T_STOP detiene el proceso
T_OK permite que el padre de este proceso realice una traza
T_GETINS devuelve un valor del espacio de instrucciones
T_GETDATA devuelve un valor del espacio de datos
T_GETUSER devuelve un valor de la tabla de procesos de usuario
T_SETINS pone un valor en el espacio de instrucciones
T_SETDATA pone un valor en el espacio de datos
T_SETUSER pone un valor en la tabla de procesos de usuario
T_RESUME finaliza la ejecución
T_EXIT realiza la salida
T_STEP activa el bit de traza

De todos estos comandos (los cuales se pasan a través de la variable *request*), los que se tratan aquí son los comandos T_OK y T_EXIT. Los comandos T_RESUME y T_STEP son parcialmente tratados aquí y completados por la tarea del sistema, que además trata el resto de los comandos.

4.2 Cabecera

En la cabecera del módulo, se incluyen las librerías necesarias para las funciones implementadas en el mismo. Se emplean, básicamente, librerías relacionadas con el manejo de memoria (mm.h), con el manejo de señales

(signal.h), la tabla de procesos (mproc.h) y los parámetros (param.h)

```
#include "mm.h"
#include <sys/ptrace.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define NIL_MPROC ((struct mproc *) 0)

FORWARD _PROTOTYPE( struct mproc *findproc, (pid_t lpid) );
```

4.3 findproc()

Esta función recibe un identificador de proceso como parámetro, y recorre secuencialmente la tabla de procesos hasta que encuentra un proceso con el identificador dado. Si lo encuentra y está activo, devuelve un puntero a la entrada de dicho proceso en la tabla; en caso contrario devuelve un cero.

4.3.1 Código

```
/*===========================================================*
 *                                findproc                                 *
 *===========================================================*/
PRIVATE struct mproc *findproc(lpid)
pid_t lpid;
{
  register struct mproc *rmp;

  for (rmp = &mproc[INIT_PROC_NR + 1]; rmp < &mproc[NR_PROCS]; rmp++)
        if (rmp->mp_flags & IN_USE && rmp->mp_pid == lpid) return(rmp);
  return(NIL_MPROC);
}
```

4.4 do_trace()

Esta función es la implementación de la llamada del sistema **ptrace()**. Se puede llamar a la misma por los siguientes motivos:

- Poner a un proceso en modo traza.
- Abortar la ejecución de un proceso.
- Reanudar la ejecución de un proceso.

Cuando llamamos a **do_trace()**, el proceso entra en modo traza. Esto no implica que se detenga su ejecución. Para detenerlo, se requiere que se ejecute

la función **kill()** con cualquier señal (salvo la SIGKILL, que mataría al proceso) y con su uid correspondiente. La llamada a **kill()** producirá una llamada a **sys_proc()**, la cual, si el proceso está en modo traza, llamará a la función de este módulo **stop_proc().**

4.4.1 Código

```
/*===========================================================*
 *                                do_trace                                   *
 *===========================================================*/
PUBLIC int do_trace()
{
  register struct mproc *child;

  /* La llamada T_OK es hecha por el hijo (creado mediante un fork)
   * del debugger antes de que ejecute el proceso a ser traceado
   */

  if (request == T_OK) {/* permite la traza por el padre de este proceso */
        mp->mp_flags |= TRACED;
        mm_out.m2_l2 = 0;
        return(OK);
  }
  if ((child = findproc(pid)) == NIL_MPROC || !(child->mp_flags & STOPPED)) {
        return(ESRCH);
  }

  /* todas las demás llamadas son hechas por el fork padre del
  * debugger para controlar la ejecución del hijo.
   */

  switch (request) {
  case T_EXIT:                  /* salida */
        mm_exit(child, (int)data);
        mm_out.m2_l2 = 0;
        return(OK);
  case T_RESUME:
  case T_STEP:                  /* finalizar la ejecución */
        if (data < 0 || data > _NSIG) return(EIO);
        if (data > 0) {         /* issue signal */
                child->mp_flags &= ~TRACED; /* la señal no es desviada */
                sig_proc(child, (int) data);
                child->mp_flags |= TRACED;
        }
        child->mp_flags &= ~STOPPED;
        break;
```

```
  }
  if (sys_trace(request, (int) (child - mproc), taddr, &data) != OK)
        return(-errno);
  mm_out.m2_l2 = data;
  return(OK);
}
```

4.5 stop_proc()

Cuando llamamos a **kill()** para un proceso en modo traza con una señal distinta que SIGKILL, la función **sys_proc()** (llamada por kill) realiza una llamada a **stop_proc().**

Esta función detiene un proceso si está en modo traza. Informa al Kernel llamando a **sys_trace(),** y si el padre estaba en estado WAITING se le envía un mensaje de respuesta (**reply())** para sacarlo de dicho estado.

4.5.1 Código

```
/*===========================================================*
 *                                stop_proc                                  *
 *===========================================================*/
PUBLIC void stop_proc(rmp, signo)
register struct mproc *rmp;
int signo;
{
/* Un proceso en modo traza recibe una señal para pararlo. */
  register struct mproc *rpmp = mproc + rmp->mp_parent;

  if (sys_trace(-1, (int) (rmp - mproc), 0L, (long *) 0) != OK) return;
  rmp->mp_flags |= STOPPED;
  if (rpmp->mp_flags & WAITING) {
        rpmp->mp_flags &= ~WAITING;      /* el padre no espera más */
        reply(rmp->mp_parent, rmp->mp_pid, 0177 | (signo << 8), NIL_PTR);
  } else {
        rmp->mp_sigstatus = signo;
  }
  return;
}
```