Proyecto ASO

Comando monitorsae

Calvo del Olmo, Álvaro
Arnillas García, Sergio
Rodrigo Ortiz, Esther

INTRODUCCIÓN

A la hora de elegir nuestro proyecto, basándonos en que era lo que podría necesitar este sistema para acercarse a sistemas operativos más modernos, encontramos adecuado realizar una monitorización del sistema, pensamos que esto podría ser muy práctico de cara al usuario.

Al principio teníamos pensado usar ficheros y usar semáforos para evitar la exclusión mútua. Como en Minix no se pueden usar semáforos había que optar por el paso de mensajes. Nos encontramos con dos problemas claves:

a) Escritura de ficheros desde el kernel. Aunque pensabamos que esto no iba a suponer un problema no fue así ya que al usar funciones como fopen o open provoca un "Kernel Panic" que interrumpe la ejecución de Minix debido a un error en los argumentos pasados en la llamada al sistema, no obstante esto es un tema que se escapa a nuestro nivel actual

b) Uso de mensajes para comunicar el kernel con el MM, y poder crear una zona de memoria compartida entre ellos. Al añadir envíos o recepciones de mensajes dentro del código se interfería con las comunicaciones que existen en el código original, por lo que se opto por otra idea: comunicar el kernel mediante una interrupción con la aplicación de usuario, y el MM mediante una syscall con la aplicación de usuario.
Finalmente no tuvimos necesidad de realizar exclusión sobre la zona de memoria compartida, ya que hasta que no se acaba de ejecutar la interrupción o la syscall no se sigue con la ejecución de la aplicación de usuario. Por lo que nunca habrá riesgos de escrituras y lecturas simultáneas.

OBJETIVO

Monitorizar el sistema operativo Minix, para informar al usuario del estado de los procesos que el sistema ha cargado.

FUNDAMENTOS TEÓRICOS

Como hemos estudiado, la estructura de Minix se encuentra dividida en 4 capas o niveles:
· Capa 1: Atiende las interrupciones, guardando y recuperando registros, planifica tareas y procesos, maneja mecanismos de mensajes,etc.
La parte de esta capa que trata con el nivel inferior de las interrupciones está escrita en ensamblador, el resto en C.
· Capa 2 - Contiene los procesos (tareas) de E/S. Todas las tareas de la capa dos y todo el código de la capa uno están combinados juntos en un único programa binario el KERNEL. Todas las tareas se gestionan independientemente y se comunican utilizando mensajes.
· Capa 3 - Contiene los servidores: el manejador de memoria, el sistema de ficheros, el servidor de red. Se ejecuta con menor privilegio que la capa 1 y 2.
· Capa 4 - Contiene todos los procesos de usuarios, Shell, editores, compiladores, etc.

Una de las mayores dificultades es la comunicación entre las capas mencionadas, por ejemplo, el kernel (capa 1 y 2) no puede comunicarse de manera directa con niveles inferiores (capas 3 y 4).
La primera opción que pensamos fue la comunicación mediante ficheros, volcando en ellos los datos para ser leídos por la aplicación. Pero la utilización de ficheros se complica al no poder utilizar funciones de alto nivel como fopen, fprintf, etc. en el kernel. Esto reducía el campo de funciones que podiamos usar a funciones de muy bajo nivel, cuya utilización era prácticamente imcompresible, por lo que optamos por el uso de interrupciones y llamadas al sistema.

Mediante una llamada al sistema es posible implementar una comunicación bidireccional entre una aplicación en modo usuario y ciertas partes del kernel, como el manejador de memoria (MM) o el sistema de ficheros (FS). Nosotros la usaremos para transmitir información del MM a nuestra aplicación "monitorsae",ya que no hemos conseguido realizar una syscall desde el kernel al MM o viceversa.
Para la comunicación entre el kernel y la aplicación se han utilizado interrupciones.

En el kernel recogemos información relativa a cada proceso. Para ellos se accede a la variable proc_addr() que devuelve la direccion del proceso pasado como argumento. Esta posición contiene información sobre el estado del proceso, su contador de programa, el puntero a pila, la memoria asignada, el estado de sus ficheros abiertos, información relativa a su planificación y a su utilización de los recursos.
El acceso a los datos de cada proceso se realiza en el MM mediante el acceso a una lista de estructuras, “struct mproc mproc[NR_PROCS]”.Los datos mas relevantes son el tamaño de los sementos de datos, pila o texto :mp_seg[X],donde “X” puede ser “D”, “S” o “T” respectivamente.

DESCRIPCIÓN DEL TRABAJO REALIZADO

Para organizar el trabajo hemos comenzando haciendo las siguientes partes por separado:

- Modificaciones en el kernel con el fin de extraer los datos deseados de la estrucutura de cada proceso, metiendo estos datos (nombre, estado y PID) en la estructura “monitor”, guardando la estructura de cada proceso en un array, “lista_monitor[]”, y mostrando la estructura al iniciar minix.
- Modificaciones en Mm obteniendo información sobre la memoria de los procesos, guardando esta también en la estructura “monitor” del mismo tipo que la definida en el kernel.
- Por otro lado hemos realizado una aplicación que mediante interrupciones fuera capaz de comunicarse con el kernel, para ser la aplicación la que mostrase los datos.

Nociones generales:

Hemos creado una aplicación llamada monitorsae que a través de una interrupción le manda un puntero al array de estructuras del tipo *"t_proceso"* previamente declarada en dicha aplicación como global. De esta forma el kernel tiene acceso a esa zona de memoria y puede rellenar los campos *"nombre"*, *"pidproc"* y *"estado"* de la estructura de cada posición del array mencionado anteriormente. Cuando la función llamada en el kernel finaliza devuelve el control a la aplicación, y ésta a continuación realiza la llamada al sistema, y le pasa la dirección del array de estructuras del tipo *"t_proceso"* al MM. Dicha syscall lanza la función *"do_memoriaprocesos"* que rellena el resto de los campos del array de estructuras (los campos referentes a la memoria). Cuando finaliza, la aplicación ya dispone de toda la información de los procesos en el array, y sólo tiene que mostrarlos por pantalla.

A continuación se explicará con más detalle cada parte:

- *Creación de una interrupción*:

Para comunicar el comando (*"monitorsae"*) la información relativa a procesos en el sistema (*estado, nombre y pid*) lo haremos mediante interrupciones. De esta manera monitorsae nos indicara en la señal (SIGUSR2) la dirección del array donde el kernel pueda escribir los datos mencionados anteriormente.

Para ello necesitamos modificar los archivos kernel/sunsighandle.c y kernel/sunsighandle.h para asociar la señal SIGUSR2 a *"capturacc"* que tratará nuestras llamadas. Estas llamadas que pueden ser 3 en principio (las que corresponderían a *send*, *receive* y *sendrec*) pero solo hicimos uso de la segunda llamada (*receive*) que modificamos y llamamos *"procesoscargados"* para que el programa de usuario, monitorsae, con solo llamarla genere la interrupción en el núcleo de Minix.

```
vectors[SIGUSR2] = capturacc;
/*Asociación de la señal SIGSR2 a la función capturacc*/
```

"procesoscargados" tiene 2 argumentos: uno donde se le pasará la dirección del array del tipo t_proc (para cargar los datos) y el otro argumento para pasarle la dirección de un entero donde el núcleo cargará un contador para que el programa de usuario (comando *"monitorsae"*) recorra cómodamente el array generado.

```
procesoscargados(&procesos,&cont_procesos);
/*llamada en monitorsae.c que generará la señal al kernel para que rellene los datos*/
```

En kernel/mpx.c definimos las funciones *"capturacc"* y *"monitorsae_sys_cal"*, donde la ultima será la que tratara las llamadas de *"monitorsae"* y quien se encargará de cargar el array con los datos.

```
void monitorsae_sys_call (int function, int src_dest, message *m_ptr, int params[3])
{
        int * puntero_cont;      /*Variable auxiliar para un puntero a entero que retornara el
                                   contador*/
        register int t;          /*Indice para recorrer los arrays*/

        register struct proc *rp;
        t_proceso * procesos;   /* Aqui se almacenara la direccion al array de t_proceso a
                                    rellenar*/
        if (params [0]==2)       /*Nos aseguramos que es la llamada correcta dentro de las
3
                                   posibles en la interrupcion SIGUSR2*/
        {
                procesos=params[1];
                for (t = 0; t <= NR_PROCS; ++t)  /*Cargamos los datos al array para el
comando
                                                   monitorsae*/
                {
                        rp = proc_addr(t);              /* t's process slot */
                        strcpy(procesos[t].nombre,rp->p_name);
                        procesos[t].estado=rp->p_flags;
                        procesos[t].pidproc=rp->p_pid;
                }
                /* Retornar el contador de procesos para que monitorsae pueda recorrer el
                array retornado*/

                puntero_cont=params[2];
                *puntero_cont=t;

        }
        else
                printf("\nERROR! monitorsae: mensaje erroneo");
        return (OK);
}
```

En la compilación del comando o programa de usuario *"monitorsae"* añadimos un archivo en ensamblador *"llamasis.s"* para que cada vez que se ejecute *"procesoscargados"* en el fichero en ensamblador se lanza la interrupción (señal) SIGUSR2, que recogerá Minix y desencadenará la ejecución *"capturacc"* y esta a su vez a *"monitorsae_sys_cal"* donde justamente ahí se realizará la tarea anteriormente descrita.

Añadir que el comando ya compilado se sitúa en /bin de la imagen del disco duro de Minix para que sea accesible fácilmente en la línea de comandos al igual que otros comandos

como *cp*, *mv*, *rm*, etc

- *Creación de una llamada al sistema*:

Mediante una llamada al sistema es posible implementar una comunicación bidireccional entre una aplicación en modo usuario y ciertas partes del kernel, como el manejador de memoria (MM) o el sistema de ficheros (FS). Nosotros la usaremos para transmitir información del MM a nuestra aplicación "monitorsae".

Pasos a seguir para añadir la llamada al sistema en el MM:

1) Crear el código de la función que queremos que se ejecute cuando lancemos la syscall. Dicho código se puede incluir en un fichero ya existente del MM para evitar tener que añadir reglas de compilación en el Makefile. Nosotros hemos añadido la función *"int do_memoriaproceso ()"* en el fichero "forkexit.c" del MM.

2) Editar el fichero "table.c" del MM para modificar la función call_vect[]. Nosotros vamos a modificar una entrada que no esté en uso y poner en ella la llamada a nuestra función. En concreto, hemos elegido la posición 0. Dicha función quedará de la siguiente forma:

```
_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
        do_memoriaproceso,      /*  0 = pruebas       */
        do_mm_exit,   /*  1 = exit      */
        do_fork,      /*  2 = fork      */
        .....
        ....
```

3) Editar el fichero "proto.h" para añadir la declaración de nuestra función do_memoriaproceso:

```
_PROTOTYPE( int do_memoriaproceso, (void) );
```

Ahora ya sólo tenemos que recompilar el MM y tendremos el sistema listo para poder atender la syscall.
A continuación se explicará la sintaxis de la syscall que se ha de ejecutar en la aplicación de usuario "monitorsae".

```
int _syscall(int m_kernel,int m_call, message *mess)
```

donde

m_kernel es el indentificador del módulo del kernel con el que queremos que atienda la syscall. Minix tiene defincas variables MM y FS para facilitar la identificación del manejador de memoria y del sistema de ficheros.

m_call es la posición donde se hace referencia a la función que queremos ejecutar con la systcall en la función call_vect[m_call] del fichero "table.c". En nuestro caso es el 0.

**mess* es un puntero a una variable tipo mensagge. De esta forma se puede pasar información al kernel. En nuestro caso, le mandamos los punteros de las variables donde queremos que el MM guarde la información de la memoria de los procesos. El MM podrá

acceder a esta información a traves de la variable “mm_in” de tipo message, definida como global.
A continuación muestro la función de nuestra aplicación que se encarga de realizar la syscall:

```
int datos_memoria()

{
message m;

  /*Para que el MM pueda recibir la posicion de las varialbes que debe rellenar*/
  m.m3_p1 = &procesos;
  m.m2_p1 = &mem_usada;

  return (_syscall(MM,0, &m)); /*Posicion 0 de la funcion do_memoriaproceso enc
vect_call[]*/
}
```

CONCLUSIONES

- Minix trata de forma independiente el kernel, MM, FS, de forma que se pueda modificar uno sin tener que hacer cambios en los demás.
- La mejor forma de comunicar el Kernel con una aplicación de usuario es usar una interrupción pasándole un puntero a la zona de memoria que se quiere compartir.
- La mejor forma de comunicar una aplicación de usuario con el MM o el FS es realizar una llamada al sistema, y al igual que en las interrupciones, mandar un puntero con la zona de memoria que se quiere compartir.
- Después de muchas pruebas, hemos concluido que para escribir ficheros desde el kernel es necesario usar funciones de bajo nivel, difícilmente comprensibles para nuestro nivel.

PROPUESTA DE AMPLIACIÓN

- Crear logs del sistema del tipo de los que crea Linux en el directorio "/proc".Sería interesante que el kernel estubiese continuamente actualizando estos ficheros, en vez de hacerlo cada vez que se ejecuta el comando.

Hemos llegado a descubrir como comunicar MM,FS y Kernel directamente a traves del paso de mensajes (cosa que a priori no se puede, ya que no ven ni ciertas variables globales), pero debido a que ya teniamos implementada otra solución y el tiempo tan ajustado no pudimos poner en practica este reciente descubrimiento en nuestra investigación.

- Ampliar nuestro comando "monitorsae" para poder bloquear/desbloquear un proceso elegido por el usuario.

BIBLIOGRAFÍA

- Universidad de Deusto**, "**Llamadas al sistema Minix" : *http://fermat.movimage.com/eside/so/data/Practicas/SO_Explicacion_Llamadas_Sistema.pdf*

- "Minix": *http://sopa.dis.ulpgc.es/ii-dso/lecminix/introduc/introduc.htm*

- "El sistema operativo Minix" http://personal.redestb.es/mick/fso/procesador/gestion%20procesador.htm

- "Ampliación de Sistemas Operativos. Planificación y Comunicación de Procesos"**:** *http://sopa.dis.ulpgc.es/ii-dso/lecminix/inicio/proc/proc_doc.pdf*

- Andrew Tanenbaum: "Operating Systems: Design and Implementation"