

# Sistemas Operativos

## 2º Curso I. T. Informática (Sistemas y Gestión) e I. Informática Práctica 5 – Adición de llamadas al sistema operativo OSO

### Consejos prácticos

En esta práctica el alumno debe modificar el sistema operativo OSO para añadirle nuevas llamadas al sistema. En este documento se describe, en primer lugar, un ejemplo muy sencillo de adición de una nueva llamada al sistema, `getpid()`, haciendo énfasis en la secuencia detallada de pasos a dar en la modificación del código. La finalidad no es otra que la de orientar al alumno en la metodología a seguir en la resolución de cada uno de los apartados de la práctica, ya que la llamada en cuestión es demasiado simple como para pretender ilustrar cualquier otra cosa. En un segundo apartado, por otro lado, se describen una serie de errores comunes a evitar durante el desarrollo de la práctica.

### 1. Adición de una nueva llamada al sistema: `getpid()`

En primer lugar, hay que tener claro el prototipo en C de la llamada en cuestión (tal y como deberá aparecer declarada en el fichero de cabecera `oso.h`, correspondiente a la biblioteca `osolib`). En este caso es el siguiente:

- `int getpid(void)`

Junto con dicho prototipo, el enunciado de la práctica nos proporciona una descripción del comportamiento deseado para dicha llamada, junto con otros detalles importantes como son el número del servicio de la interrupción 22h que será asignado a la misma (contenido en AH), así como sus parámetros de entrada y salida. En nuestro caso dicha especificación es la siguiente:

- Descripción: devuelve el PID del proceso invocador.
- Servicio: AH=03h.
- Entrada: ninguna.
- Salida: la función devuelve en AX el PID del proceso invocador.

A continuación se enumeran los pasos a dar para la solución a dicho apartado:

**Paso 1:** En primer lugar, claro está, habrá que programar la funcionalidad adecuada a la llamada en el fichero `llamadas.c`. Éste es realmente el trabajo más importante a realizar, y por tanto el menos rutinario y más dependiente de la llamada concreta (como veremos, el resto de pasos son siempre muy parecidos en todas las llamadas, y realmente mecánicos).

En el caso de ejemplo que nos ocupa, este trabajo se va a limitar simplemente a crear una nueva función `static int sys_getpid(void)` (análoga a las funciones `sys_crear_proceso(void)`, `sys_escribir(void)` y `sys_leer(void)`, ya existentes para la implementación de las tres llamadas de las que dispone OSO inicialmente). Para el `getpid()`, se trata simplemente de añadir el siguiente código en `llamadas.c`:

```
static int sys_getpid(void)
{
    return current->pid
}
```

No se nos debe olvidar añadir también en dicho fichero la nueva entrada a la tabla de llamadas:

```
struct syscall tabla_llamadas[] =
{ ..., /* Llamadas que ya había implementadas... */
  sys_getpid
};
```

Se trata de una modificación mínima, puesto que la llamada `getpid()` simplemente debe devolver el PID del proceso llamante, al que el núcleo tiene acceso a través del puntero `current`, que apunta siempre al PCB del proceso que actualmente tiene la CPU (el mismo proceso que realizó la llamada al sistema mediante la interrupción 22h). Obviamente, en un caso de llamada más compleja habría que realizar un trabajo más complicado, probablemente incluso añadiendo funcionalidad extra a algún otro módulo, y llamando a esas posibles nuevas funciones desde aquí.

Por poner un ejemplo para esto último, imaginemos que tuviésemos que implementar una llamada que tuviese que acceder al disquete para hacer algo (por ejemplo, aumentar la funcionalidad del `sys_leer()` para que lea de un fichero). Por supuesto, en este caso habría que retocar el módulo `fat.c`, dentro del cual añadiríamos nuevas funciones a las cuales llamar desde `llamadas.c`. En otros casos de distinta naturaleza (por ejemplo, para añadir la llamada `exit()`), es posible que hubiese que añadir nuevas funciones a otros módulos distintos del kernel (en el ejemplo del `exit()`, seguramente en `procesos.c`).

En cualquiera de estos casos en los que hay que tocar algún otro módulo del kernel, habría que seguir la siguiente secuencia de pasos adicionales:

**Subpaso 1.1:** Implementar la función que sea dentro del módulo `.c` pertinente (`procesos.c`, `fat.c`, o el que sea).

**Subpaso 1.2:** Declarar el prototipo de dicha función en el correspondiente fichero `.h` (`procesos.h`, `fat.h`, o el que corresponda)<sup>1</sup>.

**Subpaso 1.3:** Incluir (si no lo está ya) dicho fichero de cabecera en `llamadas.c`.

**Subpaso 1.4:** Y finalmente usar la nueva función implementada desde la función `sys_lo_que_sea()` que estamos implementando en `llamadas.c`.

Decir, por último, que la llamada `getpid()` es extremadamente sencilla, y no recibe ningún parámetro de entrada, ni escribe otro parámetro de salida (a través registros de la CPU) que no sea el propio valor devuelto por la función (que será colocado en AX por la propia función `sai_llamadas()` en `sais.c`). Para el caso más general de que se reciban parámetros adicionales (p.e., en los registros ES, BX, CX y DX, para la llamada `read`), y/o se tengan que devolver valores adicionales en algún otro registro aparte de AX (p.e., DX en la llamada `seek()`), habrá que acceder al contexto del proceso llamante (que se encuentra congelado en su pila, apuntado por `current->contexto->registro_que_sea`). Dicho acceso será para leer el valor de los parámetros de entrada, al principio de la función `sys_lo_que_sea()`, y/o para escribir los valores devueltos al final de dicha función justo antes de volver, en su caso. La función ya programada

<sup>1</sup>Obviamente, las únicas funciones de las que tenemos que incluir el prototipo en el fichero de cabecera correspondiente son aquellas a las que vamos a llamar desde fuera del módulo. Es decir, si para implementar la nueva funcionalidad que nos hace falta necesitamos apoyarnos en alguna función auxiliar, que no va a ser llamada desde fuera del módulo correspondiente, entonces no es necesario incluir su prototipo en el fichero `.h` correspondiente.

`sys_escribir()` es un buen ejemplo que ilustra este tipo de acceso al contexto del proceso llamante, que tendremos que imitar a la hora de implementar nuestras nuevas llamadas.

Una vez terminado el paso 1 ya hemos hecho prácticamente todo el trabajo importante, y ahora sólo queda retocar unos cuantos ficheros más, siguiendo una metodología que va a ser bastante mecánica y similar en todas las nuevas llamadas implementadas, como veremos a continuación.

**Paso 2:** En particular, el segundo paso va a ser simplemente retocar el fichero `llamadas.h`, para incrementar la constante `NR_LLAMADAS` con el número de llamadas implementadas. En nuestro caso, este valor pasará de valer 3 a valer 4, puesto que la llamada `getpid()` es completamente nueva<sup>2</sup>:

```
#define NR_LLAMADAS 4
```

**Paso 3:** Una vez modificado en núcleo del sistema para que ofrezca el nuevo servicio, debemos añadirlo a la biblioteca `oso.lib`. Para ello, habrá que crear un nuevo fichero `getpid.c` en el directorio `aplis/osolib` (al estilo de los ya existentes `read.c`, `write.c` y `creaproc.c`), que ofrezca un interfaz C cómodo para el usuario:

```
#include "oso.h"
int getpid (void)
{
    int valor;
    asm {
        mov ah, 3
        int 22h
    }
    valor = _AX;
    return valor;
}
```

De nuevo, al no haber parámetros de entrada, en nuestro caso simplemente se trata de inicializar el número de servicio solicitado a la interrupción 22h en el registro AH, y devolver el valor que dicho servicio devuelve en AX. Para un caso algo más general, con paso de parámetros, puede consultarse el fichero `read.c`, por ejemplo, en el mismo directorio `aplis/osolib`.

**Paso 4:** Por supuesto, el prototipo de la función anterior debe ser añadido también en el fichero `oso.h`:

```
int getpid (void);
```

**Paso 5:** Al haber incluido nuevos ficheros en la compilación, es necesario cambiar el `Makefile` correspondiente a la biblioteca `oso.lib`:

```
[...]
oso.lib: creaproc.obj read.obj write.obj getpid.obj
        tlib oso.lib ++ creaproc ++ read ++ write ++ getpid
[...]
getpid.obj: getpid.c
        tcc -mt! -c getpid.c
[...]
```

---

<sup>2</sup>Si no fuera así (en algunos apartados de la práctica no se pide añadir una nueva llamada, sino aumentar la funcionalidad de alguna ya existente, por ejemplo el `read()` para que lea también de ficheros del disquete), entonces este paso no sería necesario.

**Paso 6:** Finalmente, claro, siempre habrá que probar si el funcionamiento de la nueva llamada es correcto. Para ello, podemos crear una nueva aplicación de prueba. En nuestro caso, en lugar de ello simplemente aumentaremos la aplicación ya existente `ascii.c` para probar el nuevo `getpid()`:

```
#include "osolib\oso.h"
#include "utillib\util.h"
[...]
void main(void) {
[...]
printf("\nGETPID=%d\n",getpid());
[...]
}
```

Es muy importante que no se nos olvide nunca incluir los ficheros de cabecera correspondientes a las bibliotecas de `oso` (`oso.h`) y de utilidades (`util.h`), puesto que en este caso estamos usando una función de cada una de ellas (`getpid()` de la primera, y `printf()` de la segunda). En caso de que se nos olvide, es mucho más que probable que el compilador no genere el código adecuadamente, y se produzca el consecuente error de ejecución.

En nuestro caso, el programa `ASCII.COM` tendrá que enlazar con las bibliotecas anteriores (antes no lo hacía, puesto que simplemente contenía un bucle que accedía directamente a la pantalla y no necesitaba ninguna función externa). Por ello, habrá también que modificar el `Makefile` en el directorio `aplis` correspondiente:

```
[...]
ascii.com: c0t.obj ascii.obj osolib\oso.lib utillib\util.lib
    tlink /t c0t.obj ascii.obj, ascii.com,, osolib\oso.lib utillib\util.lib
[...]
```

Una vez llevados a cabo todos estos pasos, e instalando el nuevo kernel y las aplicaciones `SHELL.COM` y `ASCII.COM` en el disquete, debería poder ejecutarse ésta última sin problemas y, aparte de ejecutar el antiguo bucle de impresión de caracteres en el centro de la pantalla, imprimir también un mensaje 'GETPID=2', correspondiente al PID del proceso llamante, `ASCII.COM` (puesto que `KERNEL.COM` será el 0 y `SHELL.COM` el 1).

## 2. Errores frecuentes en la programación de OSO

En esta sección recogemos brevemente algunos de los errores más comunes que cometen los programadores noveles a la hora de aumentar la funcionalidad de OSO, a modo de ayuda para que los alumnos se orienten cuando "las cosas empiezan a no funcionar":

**Error 1:** Es fundamental que se conozca el prototipo de una función antes de su uso. Si no se hace, el compilador hace algunas suposiciones (la función devuelve `int` y tiene parámetros `int`) que no suelen ser adecuadas en la mayoría de los casos. Así, una fuente de errores muy habitual es no incluir los ficheros de cabecera correspondientes en los módulos o programas de aplicación que usan funciones externas, o que simplemente, aunque la función esté en el mismo módulo, es utilizada antes de su declaración<sup>3</sup>.

---

<sup>3</sup>Para este último caso, la solución es simplemente poner las funciones auxiliares delante del resto del código que las utiliza.

**Error 2:** Si para la depuración dentro del núcleo se usa `kputs()`, `kcadvalor()`, o similar (algo muy habitual, puesto que resulta de gran ayuda para el *tracado* del código cuando se está desarrollando), hay que asegurarse de que el proceso actual esté en estado `SYSCALL` para que no se le quite la CPU. Esto es necesario porque `kputs()` hace uso de la BIOS y, tal y como se explica en la documentación de OSO, la BIOS puede habilitar las interrupciones lo que, a su vez, hace que se ejecute `sai_timer()` y que, por tanto, nos puedan quitar la CPU.

Una alternativa para depurar es escribir un carácter directamente en pantalla de vídeo mediante una instrucción similar a la siguiente:

```
if(condicion)
    *((char far*)(0xB800XXXX)) = 'Y'
```

Siendo `B800:XXXX` cualquier posición de la pantalla, e `'Y'` cualquier carácter. Esta instrucción imprimirá el carácter si se cumple una condición determinada, y la podemos poner tanto en las aplicaciones para probar las llamadas (antes y después de la misma) como en los procedimientos de interfaz (antes y después de la instrucción `int 22h`), o dentro del propio kernel (antes del salto a la llamada, después de recoger los parámetros, en la propia implementación y/o antes de finalizar). También se puede poner en cualquier otra parte del kernel (SAIs de interrupciones) puesto que lo único que hace es mover un byte a memoria de vídeo. De esa forma podemos poner *espías* en diferentes partes del kernel y también podemos seguir a la CPU durante todo el recorrido de una llamada al sistema (desde que sale de la aplicación hasta que se regresa a ella) sin interferir con nada.

**Error 3:** Hay que tener extremo cuidado en el uso de los punteros. En muchas ocasiones, un puntero es utilizado desde el núcleo para apuntar a una zona fuera del segmento de éste (i.e., a la memoria de un proceso de usuario). Un ejemplo muy claro es cuando las llamadas `read()` o `write()` han de acceder (desde el núcleo) al buffer proporcionado por el usuario (en su espacio de memoria correspondiente). Naturalmente, en estos casos es absolutamente necesario que el puntero sea declarado de tipo `far`.

Un ejemplo contrario, en el que no hace falta que los punteros sean `far`, sería el de las rutinas de biblioteca `strcat(char *d, char *s)` y similares, en las que, obviamente, tanto la cadena fuente `s` como la cadena destino `d` apuntarán a zonas de memoria del propio proceso de usuario que llama a dicha función. Puesto que tanto el código como los datos implicados en la operación estarán todos en el mismo segmento, se anula completamente la necesidad de dicho tipo de punteros lejanos.

**Error 4:** Es habitual que para declarar dos punteros `far` los alumnos pongan:

```
char far * s, * t;
```

Pero este código es erróneo, ya que de hacerlo así el puntero `t` no será lejano. Lo correcto, si se quiere que `s` y `t` sean ambos lejanos, es lo siguiente:

```
char far * s, far * t;
```

**Error 5:** Es posible que durante la realización de las prácticas se tengan que realizar operaciones aritméticas como la multiplicación, la división entera o el cálculo del módulo con datos de tipo `long`. Un caso típico es trabajar con el puntero de lectura actual de un fichero, sobre el que es habitual hacer la división entera por 512 para ver el cluster correspondiente, o el módulo para hallar el desplazamiento dentro del cluster. En ese caso, puesto que el modo real de las CPUs de Intel no incluye éstas operaciones con datos de 32 bits, el compilador se apoya en rutinas auxiliares de ensamblador que, junto con otras cosas, mete en los cargadores `c0t.obj` que precisamente nosotros hemos sustituido y simplificado al máximo (ver fichero `c0t_krn1.asm` de OSO). Así, al menos en principio no es posible utilizar este tipo de operaciones

en las prácticas (puesto que, de hacerlo, el enlazador protestaría diciendo que no encuentra las subrutinas correspondientes):

```
long a,b;
b = a % 512;
b = a / 512; /* Operaciones no permitidas. */
```

En su lugar, se propone la alternativa de trabajar a nivel de bits, mediante máscaras y desplazamientos, que pueden hacer un trabajo equivalente de forma muy sencilla (aprovechándonos en este caso del hecho de que las operaciones que vamos a necesitar serán divisiones y módulos enteros por potencias de 2). A modo de ejemplo, las anteriores operaciones se podrían hacer así:

```
long a,b;
b = a & 0x000001FF; /* Módulo 512 */

[...]

long a,b;
int i;
b = a;
for(i=0;i<9;i++) { /* División por 2^9 = 512; (NO VALE b = a >> 9); */
    b = b >> 1;
}
```

Obsérvese que el desplazamiento se hace bit a bit, y no los 9 bits de golpe, como podría parecer más lógico; el problema es que el desplazamiento bit a bit de un long sí que lo puede traducir Turbo-C directamente a instrucciones en ensamblador, mientras que el de varios bits a la vez también lo hace apoyándose en una función auxiliar contenida en el `c0t.obj` original, lo que nos impide usarla en el código de OSO.

**Error 6:** Otro tipo de error habitual es que el tamaño del núcleo generado (`KERNEL.COM`) sea mayor que 8 KB y que, por tanto, el programa cargador (preparado sólo para leer 16 sectores) deje de funcionar correctamente. En ese caso, simplemente hay que incrementar adecuadamente la constante `SECTORES_SO` en los ficheros `cargador.c` y `kernel.h` (aunque realmente en este último fichero no tiene importancia), de forma que tome un valor suficientemente grande para albergar al núcleo completo (inicialmente, dicho valor es 16 porque  $16 \times 512 \text{ bytes} = 8 \text{ KB}$ ).

**Error 7:** Es también habitual que, al trabajar sobre el disquete, el fichero `KERNEL.COM` pueda no quedarnos completamente contiguo y al principio del mismo (tal y como espera el programa cargador) porque no se copió sobre él cuando éste estaba vacío. Esto se evita de forma muy sencilla asegurándonos de que el disquete esté vacío cuando copiamos sobre él dicho fichero. Sólo con posterioridad a esta operación podremos copiar en él el resto de programas y/o datos necesarios.

**Error 8:** Dado que todo el sistema depende de un compilador antiguo (Turbo C) que sólo entiende nombres de ficheros MSDOS, todos nuestros ficheros fuente deberán tener la tradicional estructura *Nombre* (máximo 8 caracteres) + *Extensión* (máximo 3 caracteres). En otro caso, el proceso de compilación podría fallar.

**Error 9:** Finalmente, también es probable que a veces se nos olvide recompilar alguna parte del código (núcleo, bibliotecas o aplicaciones) cuando hemos hecho modificado algo. Una solución relativamente cómoda es crear un *script* que haga todo el trabajo de recompilación y copia en el disquete de los ficheros involucrados con un sólo comando. He aquí un posible *script* `COMPTODO.BAT` a modo de ejemplo:

```

del /q a:*. *
cd mkboot
make
mkboot cargador.com
cd ..\kernel
make
copy kernel.com a:
cd ..\aplis\osolib
make
cd ..\utillib
make
cd ..
make
copy *.com a:
cd ..
REM Si quiero también copiar ficheros de texto de prueba:
copy *.txt a:

```

Si queremos limpiar todo el árbol de código para posteriormente recompilar todo partiendo de cero, entonces puede sernos útil este otro script, LIMPTODO.BAT:

```

del /q a:*. *
cd mkboot
del *.com *.exe *.lib *.map *.obj
cd ..\kernel
del *.com *.exe *.lib *.map *.obj
cd ..\aplis\osolib
del *.com *.exe *.lib *.map *.obj
cd ..\utillib
del *.com *.exe *.lib *.map *.obj
cd ..
del *.com *.exe *.lib *.map *.obj
cd ..

```

Tal y como están escritos, ambos scripts deberían estar ubicados en la raíz del subárbol de código de OSO.