Introducción a Sistemas Operativos: Ficheros

Clips xxx
Francisco J Ballesteros

1. Ficheros proyectados en memoria.

Utilizar `read` y `write` no es la única forma de utilizar un fichero. La abstracción fichero en UNIX tiene una relación muy estrecha con la abstracción *segmento de memoria*.

Ya sabemos que un proceso tiene diversos segmentos y sabemos que el segmento de texto procede del fichero ejecutable y se *pagina en demanda*. Recordemos que un segmento en UNIX es una abstracción. Posee una dirección de memoria virtual de comienzo, un tamaño, unos permisos (leer, escribir, ejecutar) y habitualmente se pagina desde un fichero. Para BSS y otros segmentos sin inicializar (que UNIX inicializa a cero) el fichero en cuestión suele ser `/dev/zero`.

Inicialmente, el segmento será simplemente una estructura de datos y no se le asigna memoria física (salvo que sea preciso utilizar parte de la memoria en realidad como en el caso de la pila). Una vez el programa ejecuta y utiliza direcciones de memoria del segmento, el hardware provoca un fallo de página (una excepción) si no existe memoria física asignada. El manejador, el kernel, asigna la memoria física y la inicializa, poniendo la traducción adecuada de memoria virtual a física (de *página* a *marco*).

Sabes que dado que no es posible tener una traducción en la tabla que utiliza el hardware (*tabla de páginas*) para cada dirección, las traducciones de memoria virtual a memoria física siempre traducen bloques de 4KiB de direcciones virtuales a bloques de 4KiB de direcciones físicas. A la memoria utilizada con direcciones virtuales es a lo que llamamos página y a la memoria utilizada con direcciones físicas (memoria de verdad) es a lo que llamamos marco de página. Tanto páginas como marcos deben comenzar en direcciones que sean un múltiplo del tamaño de página (normalmente 4KiB, como hemos supuesto antes en este mismo párrafo). Pero todo esto ya lo conoces de arquitectura de computadores.

Pues bien, si consideras que los ficheros (aunque procedan de disco) utilizan una cache de bloques de disco, y piensas que UNIX puede hacer que la cache para los datos del fichero sea en realidad memoria física que puede asignarse a memoria virtual... ¡Ya lo tienes!

Es posible pedir a UNIX que un segmento de memoria virtual corresponda a un fichero existente en disco. A partir de dicho momento, leer direcciones del segmento implica leer los datos del fichero (igual que se lee el código del programa conforme ejecuta). Del mismo modo, escribir direcciones del segmento implica escribir en la cache del fichero en memoria. Más adelante UNIX sincronizará el contenido del disco.

La llamada al sistema que consigue este efecto es *mmap(2)*. Tiene un número significativo de parámetros:

```
void *
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Para usarla bien, debes pensar en lo que hemos discutido antes. UNIX mantendrá páginas de memoria virtual y hará que correspondan a trozos del mismo tamaño en el fichero. La primera dirección virtual será `addr`, o bien UNIX elige una si `addr` es `0`. En total se proyectan `len` bytes desde el fichero a memoria. Como comprenderás, lo mejor será que `len` sea un múltiplo del tamaño de página. El fichero está

identificado por el descriptor `fd` y los bytes proyectados comienzan en `offset` dentro del fichero. Los parámetros `prot` y `flags` sirven para indicar los permisos para la memoria proyectada e indicar a unix propiedades que queremos para el nuevo segmento de memoria.

Por ejemplo, algunos valores interesantes para `flags` son

- `MAP_PRIVATE`: Las escrituras hechas por el proceso no se verán ni en el fichero y en la proyecciones a memoria que otros procesos hagan. Básicamente las páginas que se modifican se copian en cuanto el proceso intenta modificarlas. A esto se le llama **copy on write** y es en realidad la técnica que utiliza UNIX para hacer creer que en un `fork` el hijo es una copia del padre. Sólo se copia lo que cambia alguno de los procesos.

- `MAP_SHARED`: Las escrituras terminan en el fichero y son compartidas con todos los demás. Esto suele ser lo habitual.

- `MAP_ANON`: En realidad queremos un nuevo segmento de memoria *anónima* (que no procede de ningún fichero).

Este programa proyecta un fichero en memoria y lo modifica.

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <err.h>
#include <string.h>

enum {KiB = 1024};


int
main(int argc, char* argv[])
{
    int fd;
    void *addr;
    char *p;

    fd = open("/tmp/afile", O_RDWR|O_CREAT, 0644);
    if (fd < 0) {
        err(1, "open: /tmp/afile");
    }
    if (ftruncate(fd, 12*KiB) < 0) {
        close(fd);
        err(1, "truncate: /tmp/afile");
    }
```

```
    addr = mmap(0, 8*KiB, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    if (addr == MAP_FAILED) {
        err(1, "mmap: /tmp/afile");
    }
    p = addr;
    strcpy(&p[2*KiB], "hi at 2k\n");
    strcpy(&p[6*KiB], "hi at 6k\n");
    if (munmap(addr, 8*KiB) < 0) {
        err(1, "munmap: /tmp/afile");
    }
    exit(0);
}
```

Observa como dejamos que UNIX determine la dirección de memoria en que ponemos el nuevo segmento (es siempre lo mejor). Además, los tamaños están elegidos para que sean múltiplos de 4KiB (que presuponemos como tamaño de página, lo que hoy día suele ser cierto en todos los sistemas). Además, cuando terminamos, llamamos a *munmap(2)* para pedir a UNIX que termine la proyección.

Si ejecutamos nuestro nuevo programa podemos ver qué tiene `/tmp/afile`:

```
unix$ map
unix$ ls -l /tmp/afile
unix$ ls -l /tmp/afile
-rw-r--r--  1 nemo  wheel  12288 Aug 27 22:07 /tmp/afile
unix$ echo '12*1024' | hoc
12288
unix$ cat /tmp/afile
hi at 2k
hi at 6k
unix$
```

Y, para ser mas precisos...

```
unix$ od -A d -c /tmp/afile
0000000   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0002048    h   i       a   t       2   k  \n  \0  \0  \0  \0  \0  \0  \0
0002064   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0006144    h   i       a   t       6   k  \n  \0  \0  \0  \0  \0  \0  \0
0006160   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0012288
```

Hemos pedido a `od` con "`-A d`" que imprima las direcciones en base 10. Como puedes ver, en el offset 2048 tenemos el string que escribió nuestro programa en su memoria. Igual sucede en el offset 6144 (6KiB) con el string escrito allí. El resto de bytes está a cero dado que nunca los hemos escrito.

Si en el futuro proyectamos el fichero de nuevo, digamos en la dirección `p`, tendremos en `&p[2*KiB]` nuestro primer string tal cual esté en el fichero. Ni que decir tiene que aunque hemos escrito y leído bytes, es memoria como todo lo demás y podríamos haber utilizado cualquier otro tipo de datos.

Con todo esto ya sabes cómo crear nuevos segmentos en tu proceso. Hace tiempo se utilizaba una llamada *sbrk(2)* para aumentar el tamaño del segmento de datos, antes de disponer de las facilidades que nos da en la actualidad la memoria virtual. La implementación de *malloc(3)* utilizaba *sbrk(2)* para pedir más memoria virtual en el segmento de datos cuando agotaba la que tenía. Hoy en día, si necesitas una cantidad ingente

de memoria virtual es mejor que pidas tu propio segmento y lo uses como te plazca. De hecho, algunos sistemas UNIX crean un segmento independiente llamado *heap* para dar servicio a *malloc(3)*. Otros usan *sbrk(2)* o bien crean un segmento con suficiente tamaño de memoria virtual. En cualquier caso, es mejor dejar la implementación de `malloc` en paz y pedir lo que necesites sin molestar a la librería de C.