PRÁCTICA 4: MEMORIA Y ENTRADA/SALIDA

v1.2.2

Objetivos de la práctica

Esta práctica tiene dos objetivos. El primero es mostrar los diferentes segmentos en los que se encuentra dividido un proceso en memoria (en este caso en Linux). Se identificarán dichas divisiones y se utilizarán un par de programas parar explorar el espacio de direcciones de un proceso y analizar sus características.

El otro objetivo es enseñar cómo funciona la entrada/salida en Unix, y se verán las llamadas a sistemas necesarias para utilizarla. Esto implicará un estudio de cómo determinar cuándo se producen llamadas a sistema y qué implicaciones tiene este hecho.

1. Espacio de direcciones de un proceso

Examinar el espacio de direcciones virtual de un proceso

Examinar el espacio de direcciones virtuales de un proceso en Unix es relativamente fácil. Esto es así porque el mapa de direcciones de cada proceso está representado como un archivo. Todos estos archivos se encuentran dentro del sistema de ficheros de la CPU, que en Linux está montado en la carpeta "/proc". Este sistema de ficheros contiene toda la información pertinente a la CPU y sirve para configurarla.

Para poder gestionar y acceder a la información de los procesos de forma ordenada, "/proc" contiene una serie de directorios cuyo nombre (siempre numérico) es el PID de cada uno de los procesos que están en ejecución. Dentro de estos directorios puede encontrarse la información del proceso en cuestión. De especial interés para esta práctica es el archivo de nombre "maps", que nos detalla el espacio de direcciones del proceso.

Por ejemplo, el archivo "/proc/1234/maps" contendrá el espacio de direcciones del proceso 1234, y podríamos ver su contenido mediante la orden "cat":

```
$ cat /proc/1234/maps
```

Además de los directorios cuyo nombre es un número, dentro de "/proc" también hay otro directorio especial, de nombre "self" que apunta siempre al proceso que se esté ejecutando en un momento dado. Es, por tanto, un enlace simbólico al proceso que accede a este archivo. Por tanto, si escribimos:

```
$ cat /proc/self/maps
```

Veremos en el terminal el espacio de direcciones del proceso "cat", que el que se está ejecutando. Al hacerlo obtendremos un resultado estructurado en forma de tabla de seis columnas:

- **Direcciones:** rango de direcciones virtuales

- **Protecciones y modo de compartición:** "r" si se puede leer, "w" si se puede escribir, "x" si se puede ejecutar, "s" si está compartido con otros procesos, y "p" si es privado. Esto es, si se comparte pero se duplican las páginas en las que se escribe.
- **Desplazamiento:** desplazamiento del segmento en el fichero, si es que el segmento ha venido de un fichero.
- **Dispositivo:** dispositivo físico de respaldo del segmento (número mayor y menor), si es que el segmento ha venido de un fichero.
- **Inode:** Número de inode de respaldo del segmento, si es que el segmento ha venido de un fichero.
- **Nombre de fichero:** Nombre del fichero respaldo del segmento, si es que el segmento ha venido de un fichero.

  Fíjate en que este espacio de direcciones no tiene por qué incluir sólo el programa originalmente invocado, sino también incluye el cargador dinámico, las bibliotecas dinámicas usadas, los catálogos de mensajes, etc. Este mapa de direcciones presenta todo aquello que el proceso pueda ver.

  Si, por ejemplo, leemos el contenido de "/proc/self/maps" con "cat", el resultado será algo similar a esto:

```
00400000-0040b000         r-xp 00000000 08:05 812115      /bin/cat
0060a000-0060b000         r--p 0000a000 08:05 812115      /bin/cat
0060b000-0060c000         rw-p 0000b000 08:05 812115      /bin/cat
00a2c000-00a4d000         rw-p 00000000 00:00 0           [heap]
7f90709d2000-7f90710b5000 r--p 00000000 08:05 6978        /usr/lib/locale/locale-archive
7f90710b5000-7f907126a000 r-xp 00000000 08:05 655692      /lib/x86_64-linux-gnu/libc-2.15.so
7f907126a000-7f9071469000 ---p 001b5000 08:05 655692      /lib/x86_64-linux-gnu/libc-2.15.so
7f9071469000-7f907146d000 r--p 001b4000 08:05 655692      /lib/x86_64-linux-gnu/libc-2.15.so
7f907146d000-7f907146f000 rw-p 001b8000 08:05 655692      /lib/x86_64-linux-gnu/libc-2.15.so
7f907146f000-7f9071474000 rw-p 00000000 00:00 0
7f9071474000-7f9071496000 r-xp 00000000 08:05 655706      /lib/x86_64-linux-gnu/ld-2.15.so
7f907167e000-7f9071681000 rw-p 00000000 00:00 0
7f9071694000-7f9071696000 rw-p 00000000 00:00 0
7f9071696000-7f9071697000 r--p 00022000 08:05 655706      /lib/x86_64-linux-gnu/ld-2.15.so
7f9071697000-7f9071699000 rw-p 00023000 08:05 655706      /lib/x86_64-linux-gnu/ld-2.15.so
7fffec01f000-7fffec040000 rw-p 00000000 00:00 0           [stack]
```

Puedes ver que la memoria accesible al proceso se divide en diferentes regiones. En este caso la primera región contiene el código del programa, con permisos de lectura y ejecución (r-xp), extraída del comienzo (dirección 00000000) del fichero "/bin/cat" (*inode* 812115 del dispositivo 08:05). La siguiente región contiene los datos de sólo lectura, mientras que la tercera los de lectura/escritura. Las sigue el espacio para *heap*, que no se encuentra respaldado en disco. Podemos comprobar que la última región pertenece a la pila.

Fíjate también en que la región 5 se corresponde a los datos de localización del sistema, que las regiones 6 a 9 cubren las bibliotecas estándar de C, y que en las regiones 11 y 14- 15 se encuentra el cargador dinámico. Este último hace posible que el programa enlace correctamente las bibliotecas que va a utilizar.

Para más información puedes consultar el manual del sistema de fichero proc:

```
$ man 5 proc
```

Determinación de direcciones virtuales

A continuación se encuentra el programa "direcciones.c", que imprime en hexadecimal la dirección de memoria de diversas partes de sí mismo:

direcciones.c:

```
#include <stdlib.h>
#include <stdio.h>
int A;
void escribe(char* texto, void *dir) {


printf("Dirección de %-4s = %10x\n", texto, (unsigned int)dir);


}
int main(void) {
int B;
int *C = malloc(0x1000);
escribe ("main", main);
escribe("A", &A);
escribe ("B", &B);
escribe ("C", C);
sleep(1000);


exit(0);


}
```

Ejercicios

1. Escribe, compila el programa y ejecútalo en segundo plano (usando el operador "&"), averigua su número de proceso (usando la orden "ps"), y visualiza el contenido del archivo "maps" que corresponda al proceso. Explica, razonadamente, en qué región (código, datos, *heap* o pila) se encuentra cada dirección y por qué.

2. Explorando el espacio de direcciones

Determinación de la accesibilidad de direcciones virtuales

El siguiente programa, "accesible.c", sirve para determinar la accesibilidad tanto en lectura como en escritura de la dirección virtual que se le pasa como parámetro. Los accesos a direcciones no disponibles, igual que antes, ocasionan una excepción SIGSEGV, que es tratada por las funciones "nolee" y "noescribe", usando "signal()".

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>

int dato;
int *dir;
long *dir2;


void nolee(int s) {
          printf(" no legible\n");
          sleep(1000);
```

```
          exit(0);
}

void noescribe(int s) {
          printf(" no escribible\n");
          sleep(1000);
          exit(0);
}

void escribe(char* texto, void *dir) {

          printf ("Dirección de %-4s = 0x%lx\n", texto,(long)dir);
}

int main(int argc, char *argv[]) {


          int A;

          //escribe("A", &A);
          escribe("main", &main);


        //dir  = (unsigned int*)&A;

          dir  = (unsigned int*)&main;

          printf ("Probando la dirección virtual 0x%lx\n", (long)dir);
          signal(SIGSEGV, nolee);
          dato = *dir;
          printf(" legible\n");
          signal(SIGSEGV, noescribe);
          *dir = dato;
          printf(" escribible\n");
          sleep(1000);
          exit(0);
}
```

Escribe, compila el programa y ejecútalo.

Ejercicios

1. Usando los conocimientos aprendidos hasta ahora y el programa "accesible.c", comprueba la accesibilidad de la zona de código y la zona de datos. Justifica los resultados que se obtienen.

3. Entrada/salida estándar

Programa simple de copia de ficheros

El proceso de entrada/salida más inmediato en un ordenador personal es la copia de ficheros. Por tanto, para entender cómo funciona la entrada/salida lo más sencillo es estudiar un programa simple que haga precisamente esta función. A continuación se encuentra su listado.

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#define TAMANO 1024
char buf[TAMANO];
```

```
static void error(char* mensaje) { write(2, mensaje, strlen(mensaje)); exit(1);

}
int main(int argc, char* argv[]) {
int leidos, escritos, origen, destino;
    if (argc!=3)

error("Error en argumentos\n");
if ((origen = open(argv[1], O_RDONLY)) < 0)

error("Error en origen\n");
if ((destino = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, 0666)) < 0)

error("Error en destino\n");
while ((leidos = read(origen, buf, TAMANO)) > 0) {

if ((escritos = write(destino, buf, leidos)) < 0) error("Error en escritura\n");

    }
    if (leidos < 0)
        error("Error en lectura\n");
    close(origen);
    close(destino);

exit(0);

}
```

Como se puede ver en el código, este programa acepta dos parámetros, el primero de los cuales será el fichero origen, y el segundo el fichero destino. Si el fichero de destino ya existía será reemplazado.

Presta atención al uso de las llamadas a sistema "open", "close", "read", "write", "error" y "exit". Observa también que el fichero destino es creado con permisos de lectura y escritura para todo el mundo, es decir, 0666 (rw-rw-rw-). Hay que tener en cuenta que "umask" ya retira los permisos según su configuración.

Tras asegurarte de que entiendes cómo funciona el programa, escríbelo, compílalo y ejecútalo, copiando un fichero de cierta longitud. Crea un archivo de texto, por ejemplo, prueba.txt con algún contenido y comprueba que se copia en otro archivo, por ejemplo, prueba_copia.txt.

Trazar las llamadas a sistema de un programa

Para averiguar qué llamadas a sistema lleva a cabo un proceso durante su ejecución se utiliza el programa "strace", que las lista en el orden en el que se han producido. Para ejecutarlo no tenemos más que escribirlo seguido del programa a monitorizar. En nuestro caso concreto queremos analizar las llamadas a sistema del programa de copia que acabamos de estudiar, así que tendrás que escribir lo siguiente:

```
strace ./copia fichero_origen fichero_destino
```

Si la salida de la ejecución es demasiado grande para poder visualizar todas las llamadas a la vez (dependerá del tamaño de fichero a copiar), puedes especificar a "strace" que quieres almacenar su salida en un fichero de texto. Para ellos se usa el parámetro "-o":

```
strace -o trazas.txt ./copia fichero_origen fichero_destino
```

Analicemos las trazas resultantes. Podemos ver que hay gran cantidad de llamadas al sistema. Las primeras, denominadas prólogo, tienen que ver con la carga en memoria del proceso y el inicio de su ejecución. Fíjate en que estas llamadas suelen tener que ver sobre todo con la carga de bibliotecas (llamada "access") y la organización de la memoria a utilizar (llamadas "mmap" y "mprotect").

Es importante recordar que cuando utilizamos una llamada a sistema en un programa no estamos haciendo directamente la llamada propiamente dicha, sino invocando una función de la biblioteca "libc", que es quién en realidad ejecuta la llamada a sistema.

Uso de funciones de alto nivel

A continuación vas a modificar el programa proporcionado para que la función que imprime mensajes de error por pantalla use una función de alto nivel en lugar de la llamada a sistema "error".

Pero, antes de hacerlo, invoca el programa de tal manera que se produzca algún error (con un número de parámetros incorrecto, especificando un fichero origen inexistente, etc.) al tiempo que usas "strace" para ver las llamadas a sistema generadas.

Después sustituye en "copia.c" la función "error" por la siguiente:

```
include <stdio.h>

static void error(char* mensaje) { fprintf(stderr, "%s", mensaje); exit(1);

}
```

Compila el programa de nuevo, ejecútalo usando "strace" otra vez, forzando que se produzca algún error, y compara las trazas obtenidas con las anteriores.

Programa de copia de ficheros de Linux

Para terminar con esta parte de la práctica, ejecuta el comando de Linux "cp" usando "strace" para ver sus trazas. Compárelas con las que obtuviste al ejecutar el programa "copia.c".

Ejercicios

1. ¿Durante la ejecución de "copia.c" sin errores, qué llamadas al sistema de las mostradas usando "strace" se corresponden a las acciones que se llevan a cabo dentro de la función "main" del programa (las relativas a la copia en sí de ficheros)? Enuméralas.

2. Al cambiar la función "error" de "copia.c", ¿han cambiado las llamadas a sistema ejecutadas por el programa? ¿Ha cambiado en algo el funcionamiento de "copia.c"?

4. Entrada/salida aleatoria

Cuando en esta parte de la práctica nos referimos a entrada/salida aleatoria, no queremos decir que se acceda a un punto *al azar* del fichero, sino a punto *cualquiera*. Dicho de otra forma: no estamos obligados a acceder al contenido de un fichero en orden secuencial.

Lectura de una posición aleatoria en un fichero

Ahora vamos a utilizar el programa "leealeatorio.c", que devuelve el contenido (en hexadecimal) de una posición arbitraria dentro de un archivo. Este programa acepta dos parámetros: el nombre del fichero que se va a leer, y el número del byte a acceder.

leealeatorio.c:

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

static void error(char* mensaje) { fprintf(stderr, "%s", mensaje); exit(1);

}

int main(int argc, char* argv[]) {
    int f;
    char c;
    off_t pos;

if (argc != 3)
error("Error en los argumentos\n");

if ((f = open(argv[1], O_RDONLY)) < 0) error("Error en el origen\n");

if (lseek(f, pos = atoi(argv[2]), SEEK_SET) < 0) error("Error en el posicionamiento\n");

    if (read(f, &c, 1) != 1)
        error("Error de lectura\n");

printf("%s[%ld]= %c (%x hex)\n", argv[1], pos, c, c);

exit(0); }
```

Ahora, además de las ya conocidas llamadas al sistema *open*, *close*, *read* y *exit*, también utilizamos una nueva, *lseek*, que sirve para cambiar el punto en el que se va a leer/escribir en un archivo.

Como en los casos anteriores, ejecuta el programa y usa "strace" para analizar las llamadas a sistema.

Ejercicios

1. Modifica el programa "leealetaorio.c" para hacer que, en lugar de leer de una posición aleatoria del fichero, escriba en ella. Este programa deberá aceptar tres parámetros: el nombre del fichero, el número del byte a cambiar, y el carácter a poner. Es necesario que sea en este orden. Guarda el nuevo programa como el archivo "ejercicio4-1.c".