Introducción a Sistemas Operativos: Procesos

Clips 20 y 21
Francisco J Ballesteros

1. Carga de programas

Vimos que cuando un programa se compila y se enlaza obtenemos un fichero ejecutable. Este fichero contiene toda la información necesaria para que el Sistema Operativo pueda ponerlo a ejecutar (convertirlo en un proceso). Hay diferentes partes del binario que contienen tipos diferentes de información (código, datos, etc.) y se denominan *secciones* del ejecutable. Además, el ejecutable suele comenzar con una estructura de datos denominada cabecera que incluye información sobre qué secciones están presentes en el ejecutable. Concretamente, dónde empiezan en el fichero, de qué tipo son y qué tamaño tienen.

Una sección del fichero contiene el texto del programa (el código). Las variables globales inicializadas están en otra sección. O, con más precisión, los bytes que corresponden a los datos inicializados están en dicha sección. En realidad el sistema no sabe *nada* respecto a qué variables hay en tu programa o qué significan dichas variables. Simplemente se guardan en el ejecutable los bytes que, una vez cargados en la memoria, dan el valor inicial a las variables que utilizas. Para variables sin inicializar, sólo es preciso guardar en la cabecera del fichero ejecutable qué tamaño en bytes es preciso para dichas variables (y a partir de qué dirección estarán cargadas en la memoria). Dado que no tienen valor inicial, o mejor dicho, dado que su valor inicial es cero (todos los bytes a cero), no es preciso guardar los ceros en el ejecutable. Con indicar el tamaño de dicha sección de memoria basta.

Habitualmente, el ejecutable contiene también una tabla de símbolos con información para depuradores y programas como *nm(1)* que indica los nombres de los símbolos en el fuente, y los nombres de los ficheros fuente y números de línea. Esta información no es para el sistema operativo, es para ti y para el depurador. Para el sistema, tu programa no tiene ningún significado en particular. Sólo tu código sabe el significado de los bytes en los datos que manipula (si son enteros que hay que sumar, o qué hay que hacer con ellos).

Ya vimos como utilizar *nm* para mostrar la información de ficheros objeto y ejecutables. Consideremos ahora el programa del siguiente listado.

[global.c]:

```
#include <stdlib.h>

char global[1 * 1024 * 1024];
int global2;
int init1 = 4;
int init2 = 3;
static int stint1;
static int stint2 = 3;

int
main(int argc, char*argv[])
{
        exit(0);
}
```

Este programa tiene declaradas varias variables globales. Por un lado, `global` y `global2` son variables globales sin inicializar. Igualmente, `stint1` es una global sin inicializar, aunque declarada *static* (con lo que no se exporta desde el fichero objeto y sólo puede usarse desde el fichero que estamos compilando). Además, tenemos variables globales llamadas `init1`, `init2` y `stinit2` que están inicializadas. Si utilizamos *nm* en el ejecutable resultante de compilar y enlazar este programa vemos lo siguiente:

```
unix$ cc global.c
unix$ nm -n a.out
00000a30 T __start
00000a30 T _start
00000c30 T atexit
00000ca4 T main
...
00201008 D __progname
00201010 D __dso_handle
00201020 D init1
00201024 D init2
00201028 d stinit2
...
00401320 A __bss_start
00401320 A _edata
00401360 b stint1
00401380 B _dl_skipnum
004014a0 B global2
004014c0 B global
005014c0 A _end
crun%
```

El comando *nm* nos informa de las direcciones en que podremos encontrar, una vez cargado en la memoria, cada uno de los símbolos del ejecutable. Como puedes vers, el código del texto del programa (las instrucciones) estarán a partir de la dirección `a30` de memoria. Esos símbolos se muestran con la letra "`T`" en la salida de *nm* para indicar que son de texto.

Las variables `init1`, `init2`, y `stinit2` estarán a partir de la dirección `201020` (en hexadecimal). Puedes ver como cuatro bytes después de `init1`, en la dirección `201024`, estará `init2`. Dado que `init1` es un *int* y que en este sistema (y con este compilador) ocupa 4 bytes, eso tiene sentido. Todas estos símbolos se muestran con la letra "`D`" (o "`d`"), indicado que corresponden a datos inicializados. Las letras

que muestra *nm* son mayúsculas cuando los símbolos se exportan desde el fichero objeto (pueden usarse desde otros ficheros objeto dentro del mismo ejecutable) y minúscula en caso contrario. Fíjate en `init2` y en `stinit2` en la salida de *nm* y en el código fuente. ¿Ves la diferencia?

Las variables `global2` y `global` se muestran con la letra "`B`", indicando que corresponden a variables sin inicializar. Para estas variables, el valor inicial será cero (todos los bytes a cero). Puedes ver cómo la variable `global` está en la dirección `004014c0` y, un Mbyte después, en la dirección `005014c0` está la siguiente variable.

El código ocupará una zona de memoria contigua y tendrá permisos para permitir la ejecución de instrucciones. Habitualmente, no tendrá permisos para permitir la escritura de esa zona de memoria. El contenido de esta memoria se inicializa leyendo los bytes del fichero ejecutable que corresponden al texto del programa. Los datos inicializados ocuparán otra zona de memoria contigua, con permisos para leer y escribir dicha memoria. Esta memoria se inicializará leyendo del fichero ejecutable los bytes que corresponden al valor inicial de esa zona de memoria. Los datos sin inicializar ocuparán otra zona, inicializada a cero.

Cada una de estas zonas es contigua, tiene una dirección de comienzo y un tamaño, y la memoria que ocupa tiene las mismas propiedades (permisos, cómo se inicializa, etc.). Es precisamente a eso a lo que se denomina *segmento*. Cuando el program ejecute, tendrá un segmento de texto con el código, otro de datos con variables inicializadas, otro denominado *BSS* con datos sin inicializar (que parten con los bytes a cero) y otro de pila. Véase la figura 1. El nombre BSS viene de una instrucción de una máquina que ya no existe que se utilizaba para inicializar memoria a cero (y son las iniciales de *Base Stack Segment*, aunque hoy día no tiene que ver con la pila).

Cuando el sistema ejecute el programa contenido en este fichero ejecutable, cargará en la memoria el código procedente del mismo, y los valores iniciales para variables inicializadas. Igualmente, la memoria que ocupen las variables sin inicializar se inicializará a cero y se creará una pila para que puedan hacerse llamadas a procedimiento. El trozo de código del kernel que se ocupa de todo esto se denomina cargador.

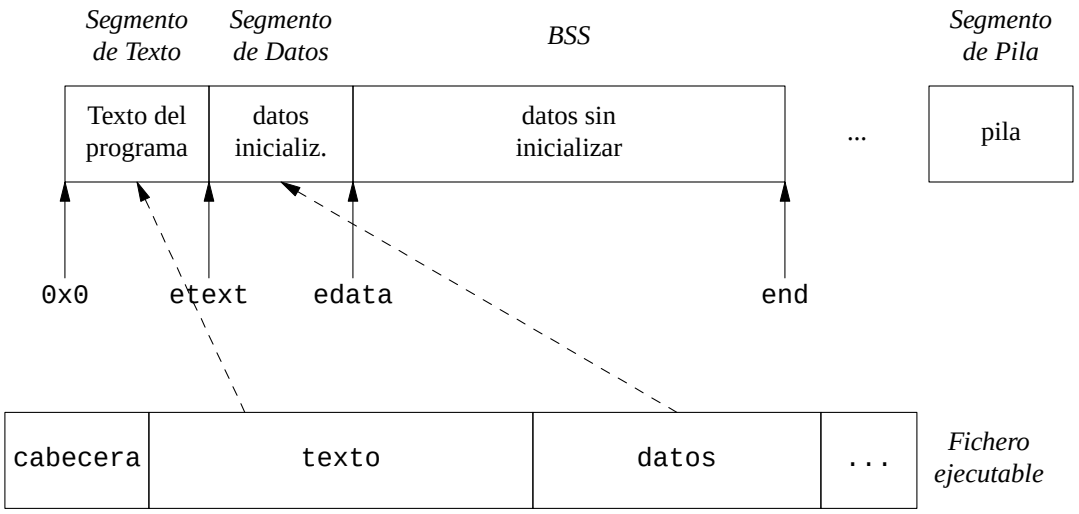

Figura 1: *El cargador se ocupa de cargar los segmentos de un proceso con datos procedentes del ejecutable.*

Pero es importante que comprendas que el sistema *no sabe nada* sobre tu programa. Ni siquiera sabe ni qué funciones has definido ni qué variables utilizas. ¡Ni le importa! Si utilizamos el comando *strip(1)* para eliminar la información de la tabla de símbolos del ejecutable, podremos verlo.

```
unix$ ls -l a.out
-rwxr-xr-x  1 nemo  wheel  8729 May 23 14:59 a.out*
unix$ strip a.out
unix$ ls -l a.out
-rwxr-xr-x  1 nemo  wheel  6776 May 23 15:26 a.out*

unix$ nm -n a.out
00000000 W __cxa_atexit
00000cc0 T __fini
00201008 D __data_start
00201008 D __progname
003012a0 D __got_start
00301320 D __got_end
00401320 A __bss_start
00401320 A _edata
005014c0 A _end
```

Tras ejecutar *strip*, *nm* no puede obtener información alguna respecto que qué variables globales o qué funciones están contenidas en el ejecutable. Como hemos compilado con enlace dinámico aún quedan algunos símbolos que son necesarios para que el enlazado dinámico funcione, pero eso es todo. ¡Nadie puede saber que había algo llamado `global` en tu código!

El cargador se limitará a cargar los bytes de texto hacia el segmento de texto y los bytes de datos inicializados hacia el segmento de datos. Lo que signifiquen esos bytes es cosa tuya. En cuanto al BSS, el ejecutable indica en la cabecera cuantos bytes son necesarios, pero naturalmente no guardamos 1Mbyte de ceros en el ejecutable para nuestra variable `global`. Simplemente se indica en el ejecutable en número total de bytes a cero y la dirección en que deben comenzar.

Volvamos a pensar en los segmentos de un proceso. En la figura 1 puedes ver que el enlazador suele dejar en el ejecutable un símbolo llamado `etext` cuya dirección es el final de texto, otro `edata` cuya dirección es el final de los datos inicializados y otro `end` cuya dirección es el final del BSS. Con estos símbolos puedes saber qué direcciones utilizan tus principales segmentos. El comando *size(1)* muestra el tamaño que tendrán los segmentos de un programa cuando lo ejecutes:

```
unix$ size a.out
text    data    bss     dec     hex
2845    500     1048992 1052337 100eb1
```

Para nuestro programa, el BSS tendrá `1048992` bytes.

Los segmentos son parte de la abstracción que te da UNIX para que tengas procesos. Como habrás observado, aunque ejecutemos muchos procesos, todos ellos piensan que tienen la memoria para ellos solos. Por ejemplo, en nuestro programa la variable `global` estaría en la dirección `004014c0`. Si lo ejecutamos varias veces de forma simultánea, cada uno de los procesos tendrá su propia versión y pensará que es el único en el mundo. En todos ellos, `global` estará en la dirección `004014c0`.

Simplemente, la memoria del proceso es memoria virtual. Esta memoria no existe en realidad. Por eso se la denomina virtual. El sistema utiliza el hardware de traducción de direcciones (la unidad de gestión de memoria o MMU) para hacer que las direcciones que utiliza el procesador se cambien por otras *al vuelo* antes de que el hardware de memoria las vea. Cada proceso utilizará zonas distintas de memoria física y, gracias a la traducción de direcciones, la dirección `004014c0` de cada proceso se cambia por la dirección de memoria física que dicho proceso utilice. Pero todos ejecutan instrucciones que acceden a `global` a partir de la dirección `004014c0`.

Las direcciones de memoria virtual se traducen a direcciones de memoria física empleando una tabla (que inicializa para cada proceso el sistema operativo) llamada *tabla de páginas*. Esta tabla traduce de 4 en 4

Kbytes (o un tamaño similar) las direcciones. A cada trozo de 4 Kbytes de memoria virtual lo llamamos página y al trozo correspondiente de memoria física lo denominamos marco de página.

La primera página de memoria suele dejarse sin traducción, lo que hace que sea imposible utilizar sus direcciones sin ocasionar un error (similar al que se produciría si divides por cero). Este error o excepción se atiende de forma similar a una interrupción y, habitualmente, el sistema operativo detiene la ejecución del programa que lo ha causado. Por eso no puedes atravesar un puntero a *nil*. La dirección de memoria cero es inválida, y *nil* es el valor cero utilizado como dirección de memoria.

Todo esto es importante puesto que tiene impacto en tus programas. Habitualmente el sistema carga tu código y datos, y te asigna memoria física, conforme utilizas direcciones de memoria virtual en tus segmentos. Inicialmente no habrá traducciones a memoria física para casi nada en tu proceso y, conforme el procesador utilice direcciones, el sistema irá cargando en demanda el resto. A esto se lo denomina *paginación en demanda*.

La paginación en demanda es fácil de implementar puesto que el sistema sabe qué segmentos tiene el proceso y qué direcciones usan estos. Además, sabe cómo inicializar la memoria de dichos segmentos (leyendo valores iniciales del fichero ejecutable o inicializándola a cero, por ejemplo). Inicialmente, un segmento puede tener sus páginas sin traducción a (marcos de página en) memoria física. Cuando el procesador utiliza una dirección en dichas páginas, el hardware eleva una excepción y el manejador instalado por el sistema operativo la atiende. Dicho manejador comprueba que en teoría el proceso debería poder utilizar dicha página de memoria y asigna un marco de página (poniendo una traducción en la tabla de páginas) inicializado según corresponda. Cuando el manejador retorna, el programa de usuario continúa su ejecución como si nunca hubiese sucedido el fallo de página.

Por ejemplo, `global` está inicializado a cero porque las páginas en que reside serán inicializadas a cero por el sistema cuando las asigne. Y será así dado que dichas páginas forman parte del BSS. Si tu proceso lee una posición de `global` por primera vez, es muy posible que al proceso se le asignen 4Kbytes de memoria física para la página a que accede, inicializados a cero. Y por eso tu proceso creerá que `global` está inicializado a cero. Antes de utilizar el array, el MByte es sólo memoria virtual. ¡No consume memoria física! Eso si, si se te ocurre la "buena" idea de utilizar un bucle para inicializar tu array a cero, tu proceso accederá a todas esas direcciones y el sistema le asignará la memoria correspondiente. La memoria virtual es gratis, la memoria física no. Es muy habitual utilizar arrays de gran tamaño sabiendo que sólo van a utilizarse unas pocas posiciones en tiempo de ejecución, por ejemplo para grandes tablas hash y para otros propósitos en algunos run-times de lenguajes de programación. ¿Comprendes por qué puede hacerse esto?

Sucede lo mismo con la pila hoy día. Hace tiempo, el segmento de pila solía crecer a medida que el proceso utilizaba más espacio en la misma. Hoy día, el segmento de pila suele tener un tamaño prefijado y no crece. Dado que la memoria que usa es *virtual*, inicialmente dicha memoria es gratis: no tiene asignada memoria física. Naturalmente, esto es así hasta que dicha memoria se usa a medida que crece la pila.