Atomik Kernel API

loader

Una de las funciones del microkernel consiste en ejecutar diversos tipos de procesos (hilos del kernel, procesos de sistema y procesos de usuario). Para el caso de los procesos de sistema y usuario, Atomik debe crear un espacio de direcciones privado, diferente al del resto del kernel (que además mapea toda la memoria física de todo el sistema), donde se cargan las páginas de código y datos en direcciones virtuales concretas de las que se compone el programa a ejecutar.

La forma más corriente de describir esta información es mediante un archivo ejecutable, como por ejemplo a.out, ELF, PE32, o Mach-O. Atomik ha sido diseñado para ser agnóstico a cualquier formato de fichero ejecutable : lo único que necesita saber es qué páginas deben cargarse en qué direcciones y con qué permisos, y dónde se encuentra su punto de entrada. Para este fin, Atomik proporciona la API **loader**, la cual ofrece un conjunto de funciones con las que se permite definir nuevos formatos de fichero ejecutable y manejarlos de forma completamente transparente.

Tipos, prototipos y macros pueden encontrarse en el fichero de cabecera `<task/loader.h>`

Estructuras

```
struct loader
{
  const char *name;
  const char *desc;

  void *        (*open)    (const void *, busword_t);
  busword_t     (*entry)   (void *);
  int           (*walkseg) (void *, struct vm_space *, int (*) (struct
vm_space *, int, int, busword_t, busword_t, const void *, busword_t));
  void          (*close)   (void *);
};
```

Esta estructura sirve para describir un formato de ejecutable a partir de las funciones necesarias para procesarlo. **Todo cargador de un formato ejecutable debe implementar las cuatro funciones apuntadas por esta estructura** (es decir, ninguna puede valer `NULL`) para poder ser utilizado por Atomik. El significado de los campos es como sigue :

`const char *name;`
: Nombre del formato. Es una cadena corta, acabada en caracter nulo y preferiblemente en minúsculas, describiendo el formato en cuestión. Por ejemplo, « `elf32` » o « `pe32` ».

`const char *desc;`

Descripción más prolija del formato. Es una frase describiendo el formato ejecutable, como por ejemplo « Executable and Linking Format – 32 bits » o « Portable Executable – 32 bits ».

`void *(*open) (const void *base, busword_t size);`

Dirección de memoria de la función de apertura del binario. Es la primera función que se va a llamar cuando se intente interpretar un binario. Esta función acepta como argumento `base` la dirección de memoria donde se encuentra el primer byte del ejecutable , y un segundo argumento `size` con el tamaño del binario. Esta función, que debe ser implementada por el intérprete del formato que se quiere soportar, debe devolver un puntero a alguna zona de memoria útil para el dicho intérprete, como por ejemplo punteros hacia tablas de cabeceras de segmento u otros metadatos. Si no se ha podido abrir (por ejemplo, no se reconoce o el ejecutable o está corrupto), la función debe devolver `KERNEL_INVALID_POINTER`.

Nótese que el puntero devuelto es una estructura opaca al resto de la API : lo único que le interesa a Atomik es que es un puntero hacia cierto estado interno que debe ser pasado como argumento a todas las funciones del intérprete.

`busword_t (*entry) (void *opaque);`

Dirección de memoria de la función que devuelve el punto de entrada del binario, es decir, dirección de memoria dentro del espacio de direcciones que el binario solicita a la que se debe saltar una vez cargado. El argumento `opaque` es el mismo puntero devuelto previamente por la función `open`.

`int (*walkseg) (void *opaque, struct vm_space *target_space, int (*segment_cb) (struct vm_space *space, int type, int flags, busword_t virt_addr, busword_t mem_size, const void *base, busword_t fil_size));`

Dirección de memoria de la función que va a recorrer todos los segmentos definidos por el fichero ejecutable. Por cada segmento, esta función debe llamar a la función `segment_cb` con los siguientes argumentos :

`space`: Puntero a la estructura codificando el espacio de direcciones objetivo. Su contenido debe considerarse opaco, y su valor debe ser el del argumento `target_space` con el que se ha llamado a `walkseg`.

`type`: Tipo de segmento que se ha detectado, descrito a través de las macros `VREGION_TYPE_[...]` definidas en `<mm/vm.h>`. Para procesos del sistema que no están asociados a ningún fichero, el tipo de segmento ha de ser `VREGION_TYPE_ANON` o `VREGION_TYPE_ANON_NOSWAP`. En ningún caso este valor puede ser `VREGION_TYPE_STACK`, ya que es el microkernel el encargado de reservar la memoria necesaria para la pila.

`flags`: Propiedades del segmento a crear. Es un valor que se obtiene aplicando la operación OR (|) entre los valores `VREGION_ACCESS_READ` (para que el segmento sea leíble), `VREGION_ACCESS_WRITE` (para que el segmento sea escribible) y `VREGION_ACCES_EXEC` (para que el segmento sea ejecutable).

`virt_addr`: Dirección virtual del inicio del segmento. Debe estar alineada a `PAGE_SIZE`.

`mem_size`: Tamaño (en memoria) del segmento a crear.

`base`: Dirección de memoria donde se encuentra el primer byte de los auténticos datos de este segmento. Es una dirección que debe calcular la implementación de `walkseg` para poder especificar desde dónde se deben copiar los datos del ejecutable al espacio de direcciones del nuevo proceso.

`fil_size`: Tamaño de los datos apuntados por `base`. No es inusual que este tamaño sea menor que `mem_size` (sobre todo en el segmento de datos). El espacio no cubierto por los datos contenidos en el ejecutable se rellena con ceros.

Nótese que `walkseg` no tiene por qué reservar memoria más allá de las necesidades impuestas por la implementación de su estado interno. Las reservas de páginas y espacios virtuales están ocultas en la función apuntada por `segment_cb`. Es más, dicha función no tiene ni por qué modificar un espacio de direcciones (podría ser simplemente una función de depuración que enumere los segmentos que un binario podría cargar).

`busword_t` `(*entry) (`**`void`** `*opaque);`

Dirección de memoria de la función encargada de liberar cualquier recurso utilizado en la apertura e interpretación del binario, cuyo estado se encuentra en `opaque`. Esta función es llamada justo antes de solicitar la liberación de los recursos reservados para abrir el binario.

Funciones

`struct loader *loader_register (const char *name, const char *desc)`

Reserva una estructura de tipo loader para definir un nuevo cargador de formato ejecutable, con el nombre `name` y una descripción `desc` y devuelve su dirección, o devuelve `KERNEL_INVALID_POINTER` en caso de error. Esta estructura es global, y como estas cadenas no son copiadas (simplemente se copian los punteros tal cual son pasados a la función), se debe a segurar que no están en la pila (p. ej, como un array de caracteres automático). Esta estructura, definida en el punto anterior, tiene inicialmente sus punteros a función inicializados a `NULL`.

Aunque esté reservada, la API no la utilizará como un formato válido hasta que todos sus campos estén rellenados con valores distintos a NULL.

Reentrante : no
Thread-safe : no

loader_handle *loader_open_exec (struct vm_space *space, const void *exec_start, busword_t exec_size)

Intenta abrir un ejecutable ubicado en un búfer de memoria exec_start y de tamaño exec_size, probando todos los intérpretes válidos registrados por la API de carga de ejecutables hasta encontrar uno que sea capaz de interpretarlo. El argumento space, que puede ser opcionalmente NULL, será pasado a la función de enumeración de segmentos (walkseg) y que podría ser usado para inicializar un espacio virtual de direcciones.

La función devuelve un puntero a loader_handle, una estructura que define el binario que se pretende interpretar, o KERNEL_INVALID_POINTER en caso de que el binario no se pudiese abrir.

Reentrante : no
Thread-safe : no

busword_t loader_get_exec_entry (loader_handle *handle)

Devuelve el punto de entrada del binario identificado por handle (previamente obtenido a través de la función loader_open_exec). La dirección devuelta es una dirección virtual respecto al propio espacio de direcciones que el binario representa.

Reentrante : sí (sólo si el intérprete subyacente también es reentrante)
Thread-safe : no

int loader_walk_exec (loader_handle *handle, int (*callback) (struct vm_space *, int, int, busword_t, busword_t, const void *, busword_t))

Recorre todos los segmentos del binario identificado por handle, llamando a la función callback por cada uno de ellos (la semántica de los argumentos esta función se puede encontrar en la definición de la estructura loader en el punto anterior). La función devuelve KERNEL_SUCESS_VALUE en caso de éxito o KERNEL_ERROR_VALUE en caso de error.

Reentrante : sí (sólo si el intérprete subyacente también es reentrante)
Thread-safe : no

```
void
loader_close_exec (loader_handle *handle)
```

Cierra el binario identificado por `handle`, liberando todos los recursos asociados a él. Se debe llamar a esta función cuando se termine de leer el fichero ejecutable para evitar fugas de memoria.

Reentrante : sí (sólo si el intérprete subyacente también es reentrante)
Thread-safe : no