Sistemas Operativos Avanzados (MUII)
Ejercicio sobre gestión de memoria (28-mayo-2021)
El largo camino del acceso simbólico al físico

Planteamiento del trabajo

La gestión de memoria es una labor global en la que no solo participan el sistema operativo y el hardware subyacente, sino que intervienen otros componentes tales como el compilador y el montador. Para entender de una forma integrada esa labor, se considera conveniente estudiar cómo se articulan todos estos componentes para alcanzar esa meta, incluso aunque algunos de ellos no forman parte del sistema operativo propiamente dicho.

El código de un programa escrito en un lenguaje de alto nivel define de forma simbólica variables y funciones y hace referencia a las mismas utilizando su nombre simbólico. En tiempo de ejecución, esas funciones y variables tendrán asignadas direcciones físicas de memoria y el programa deberá generar accesos a memoria que correspondan a esas direcciones físicas. Los distintos componentes del sistema de gestión de memoria visto de una manera integral (compilador, montador, sistema operativo y MMU) realizarán todo este proceso, cuyo estudio es el objetivo de este trabajo.

A priori, puede parecer que este proceso de traducción y correspondencia entre el acceso simbólico y el real es bastante directo: el sistema de compilación asigna una dirección de memoria a cada símbolo, y resuelve cada referencia a ese símbolo de manera que se convierta en tiempo de ejecución en un acceso a dicha dirección. En un escenario donde un programa estuviera formado por un único fichero de código fuente y ejecutase en un sistema monoprogramado con un procesador sin MMU, las cosas serían básicamente así. Sin embargo, en un sistema de propósito general las circunstancias son distintas:

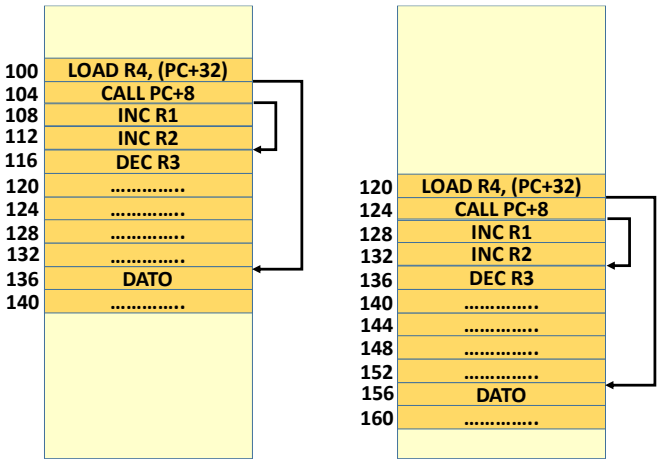
- Un programa puede estar formado por varios ficheros de código fuente. El compilador procesa de forma independiente cada uno de ellos realizando una primera asignación de direcciones en el ámbito de ese módulo, así como una resolución de las referencias vinculadas con símbolos definidos en el mismo módulo.
- En una fase posterior, el montador procesa conjuntamente los ficheros objeto del programa agrupando las secciones del mismo tipo en regiones o segmentos y generando un fichero ejecutable que tiene su propio espacio lógico. Este agrupamiento conlleva la reubicación de las direcciones asignadas a algunos símbolos y de las referencias a los mismos, así como la resolución de las referencias a símbolos definidos en otros módulos. Al final de esta etapa, en el ejecutable los símbolos tienen asignadas direcciones lógicas y las referencias a estos símbolos quedan vinculadas a esas direcciones lógicas.
- En tiempo de ejecución, el sistema operativo asigna direcciones físicas a las direcciones lógicas que genera el proceso construyendo para cada proceso las tablas de páginas que reflejan esta asignación. La MMU usa las tablas de páginas del proceso activo para realizar esta traducción.

En consecuencia, ese proceso de asignación de direcciones a los distintos objetos y de resolución de los accesos simbólicos a los mismos no es tan directo y conlleva un camino de transformaciones algo más largo y tortuoso, como se apreciará en este ejercicio.

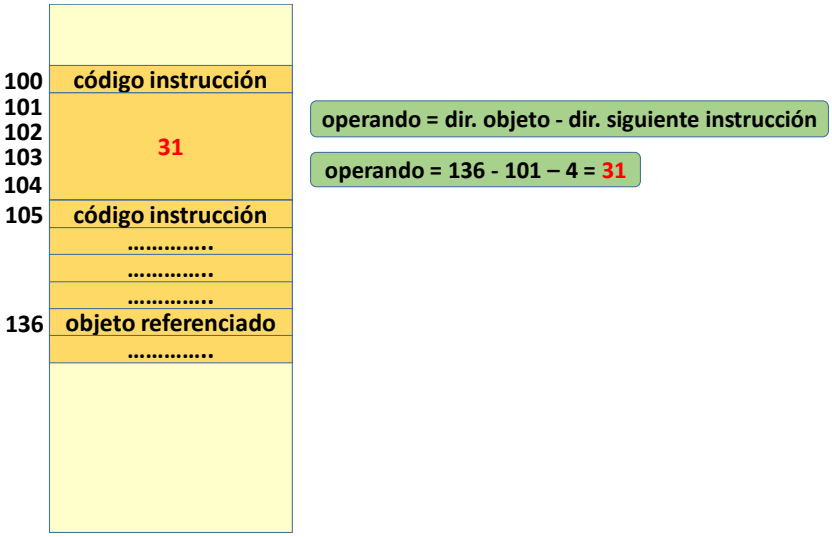
Los experimentos planteados en este trabajo se basan en los que se incluyen en el capítulo de gestión de memoria (capítulo 6) del primer volumen del libro de texto recomendado para la asignatura y que vienen identificados en ese capítulo precisamente bajo el epígrafe de experimentos. En cualquier caso, en ese libro se profundiza mucho más en todo este proceso, estudiando con detalle el formato de ejecutable ELF.

Hay que resaltar que los resultados obtenidos en los experimentos de este ejercicio pueden cambiar de forma significativa dependiendo de la versión de la plataforma usada, aunque los conceptos son básicamente los mismos. Para el planteamiento del ejercicio, se ha usado una distribución de Linux Ubuntu 20.04 con un núcleo 5.8.0 y una versión del compilador 9.3.0 para un procesador x86 de 64 bits.

Antes de entrar en el detalle del ejercicio, se considera conveniente recordar que este procesador usa normalmente un direccionamiento relativo al contador de programa tanto para las referencias a datos como para los saltos y llamadas a función. Ese esquema facilita la generación de código independiente de la posición (PIC: *Position Independent Code*), como ilustra la siguiente figura que usa un código ensamblador hipotético, donde se puede apreciar cómo, gracias al uso de un direccionamiento relativo al contador de programa, tanto para las referencias a datos como para las llamadas, el código puede ejecutar en cualquier posición de la memoria sin necesidad de realizar ningún tipo de modificación para reubicarlo.



Hay que aclarar que, como se aprecia en la siguiente figura y en la previa, para calcular qué desplazamiento especificar en el operando de una instrucción que usa direccionamiento relativo al contador de programa, se debe tener en cuenta que durante la ejecución de una instrucción el contador de programa apunta ya a la siguiente. Por tanto, ese desplazamiento corresponderá a la distancia en memoria entre el objeto y la dirección donde se almacena la siguiente instrucción. Nótese que en la figura se ha calculado la dirección de la siguiente instrucción como la distancia entre el objeto referenciado y la suma de la dirección del propio operando que contiene la referencia (101) más un cierto desplazamiento (4) que corresponde a la distancia entre ese operando y el inicio de la siguiente instrucción. El sistema de compilación y montaje usa habitualmente este esquema.



A continuación, se analizan las distintas fases que conlleva el ciclo de vida de un programa: la compilación, el montaje y la ejecución.

Experimento de compilación

Vamos a tomar como ejemplo el fichero *mod.c*, incluido en el directorio *programas* del material de apoyo proporcionado (*memoria.tgz*), que incluye definiciones de distintos tipos de símbolos, así como referencias a los mismos. Concretamente, define 5 símbolos: una variable estática global, una variable estática de módulo, una variable local no estática (nótese que se pueden definir variables de función estáticas usando el calificador *static*), una función de módulo y una función global. Todas ellas son referenciadas desde el propio módulo, exceptuando la función global, que será invocada solo desde el módulo que contiene la función *main*. A continuación, se especifican los distintos pasos de este experimento:

- Antes de comenzar los experimentos, compile (*make*) los programas del material de apoyo (directorio *programas* de *memoria.tgz*).
- Imprima los símbolos usados en el objeto (podría haberse usado *readelf -s*, pero *nm* es más sencillo; recuerde que en Linux se utiliza el formato ELF, que es aplicable tanto a ficheros objeto como a ejecutables y bibliotecas dinámicas), es decir, la tabla de símbolos:

  nm mod.o

  En la salida impresa por este mandato, hay una línea por cada símbolo, tal que en cada línea aparece en la primera columna la dirección asignada al símbolo, la sección en la segunda (T código, R constantes, D datos con valor inicial y B datos sin valor inicial; U índica que el símbolo no está definido en este módulo), donde un carácter en minúsculas indica que es un objeto local al módulo y en mayúsculas que es global, y en la tercera columna el nombre del símbolo.

- Obtenga el código ensamblador de ese módulo (recuerde que el registro *rip* es el contador de programa y *rbp* el puntero al marco de pila que se usa en los accesos a parámetros y variables locales) y fíjese especialmente en el correspondiente a las referencias a los símbolos (nótese que la operación de incremento genera dos referencias al símbolo: una para traerlo de memoria a un registro y otra para volver a llevarlo a memoria una vez incrementado):

  objdump -S -Mintel mod.o

- Acceda a la tabla de reubicaciones del módulo, fijándose únicamente en la sección denominada *.rela.text*, que incluye la información de las reubicaciones y resoluciones que requiere el código (sección *.text*) del módulo.

  readelf -r mod.o # solo nos interesa .rela.text que aparece al principio del listado

  A continuación, se explica con más detalle la información mostrada por este mandato.

En la tabla de reubicaciones de un fichero objeto se almacena información sobre dos tipos de referencias:

- Referencias a símbolos definidos en el propio módulo, pero que es necesario reubicar en tiempo de montaje. En tiempo de compilación, se asignan direcciones a los símbolos, pero en el contexto de la sección correspondiente de ese módulo. Así, por ejemplo, a la primera variable estática con valor inicial se le asigna la dirección 0 dentro de la sección *.data* (así se denomina la sección de datos estáticos con valor inicial; la de código se denomina *.text*, la de constantes *.rodata* y la de datos sin valor inicial *.bss*). Si esa variable ocupa 4 bytes, la siguiente variable del mismo tipo encontrada tendrá asignada la dirección 4 y así sucesivamente. En tiempo de montaje, como se analiza en la siguiente sección, se agrupan las secciones de los diversos objetos del programa reasignando a cada símbolo estático una dirección en el ámbito del espacio lógico del ejecutable. Así, si el símbolo en el objeto

tiene asignada la dirección *X* de la sección *.data* y en tiempo de montaje esa sección *.data* de ese objeto queda ubicada, como parte del agrupamiento de secciones, a partir de la dirección lógica *Y*, en tiempo de montaje la dirección lógica asignada a ese símbolo es *X+Y* y deben ajustarse todas las referencias a ese símbolo para que contengan ese valor. En consecuencia, en la entrada de la tabla de reubicaciones correspondiente a esta referencia, hay que guardar el identificador de la sección que contiene el símbolo referenciado, así como la dirección del mismo dentro de esa sección para poder hacer la reubicación en tiempo de montaje.

- Referencias a símbolos no resueltos al no estar definidos en este módulo. Hay que guardar el nombre del símbolo en la entrada de la tabla correspondiente a esta referencia para ser resuelta en tiempo de montaje.

Hay distintas maneras de gestionar esta información de reubicación. En la plataforma usada para preparar este ejercicio se usa la modalidad RELA en la que el operando de la instrucción que realiza la referencia almacena un 0 y en la entrada de la tabla de reubicaciones correspondiente a esa referencia, como se ha explicado previamente, se guarda o bien el identificador de la sección y la dirección del símbolo dentro de la misma, en el caso de una referencia resuelta, o bien el identificador del símbolo, si se trata de una referencia no resuelta. Dado el uso en x86_64 de direccionamiento relativo al contador de programa para todas las referencias, se almacena también en esta entrada, tal como se explicó en la figura previa, un reajuste que corresponde a la distancia entre la dirección del operando que contiene la referencia y de la siguiente instrucción (normalmente, el reajuste tiene un valor de -4 porque el operando ocupa 4 bytes y aparece en la parte final de la instrucción). Así, en el caso de la referencia a un símbolo no resuelto, en la entrada de la tabla aparecerá el identificador del símbolo y un -4, mientras que en el caso de una referencia resuelta pero que hay que reubicar en tiempo de montaje, aparecerá el identificador de la sección y la suma de la dirección del símbolo dentro de la sección y el -4. En el ejercicio sobre el montaje, completaremos el análisis de este mecanismo.

De la información mostrada por el mandato *readelf -r*, que imprime una línea por cada referencia que debe procesarse en tiempo de montaje, estamos interesados en las siguientes columnas:

- La primera columna corresponde al desplazamiento dentro de la sección de código del operando de la instrucción que hace la referencia. Al finalizar el montaje, ese operando deberá contener, directa o indirectamente, la dirección del símbolo referenciado.
- La tercera columna muestra el tipo de reubicación. Por ejemplo, la modalidad R_X86_64_PC32 indica que se trata de una referencia relativa al contador de programa con un desplazamiento de 32 bits en un sistema x86-64.
- El valor mostrado en la última columna depende de si se trata de una referencia resuelta, en cuyo caso se incluye el identificador de la sección y la dirección del símbolo dentro de la sección, o no resuelta, donde se almacena el identificador del símbolo. Recuerde que, en ambos casos, aparece también el reajuste requerido por el direccionamiento relativo al contador de programa y que, en el caso de una referencia ya resuelta, en la salida se muestra la suma de la dirección del símbolo y el reajuste.

Además del fichero *mod.c*, el programa contiene el módulo *main.c* que incluye la función *main* y simplemente referencia los dos símbolos globales definidos por el otro módulo.

- Revise el código ensamblador y la tabla de reubicaciones del segundo módulo,:

  objdump -S -Mintel main.o

  readelf -r main.o # solo nos interesa .rela.text que aparece al principio del listado

Nótese que en este ejercicio nos centramos en las referencias a símbolos desde el código (por eso la sección se denomina *.rela.text*), pero también puede haber referencias desde los propios datos (por ejemplo, una

variable global de tipo puntero que tiene como valor inicial la dirección de otra variable global). Esas referencias están en otra tabla de reubicaciones.

Por último, vamos a resaltar dos aspectos relacionados con todo este proceso de resolución de símbolos:

- Observe que en el primer módulo no hay referencias no resueltas ya que todas corresponden a símbolos definidos en el propio módulo. Sin embargo, las referencias a símbolos globales definidos en el propio módulo se tratan de la misma manera que las referencias no resueltas ya que la resolución definitiva de estas referencias puede decidirse incluso en tiempo de ejecución y corresponder finalmente a un símbolo de una biblioteca dinámica en lugar de al símbolo definido en el módulo.
- En cuanto a las referencias desde el código a símbolos estáticos de módulo, se resuelven en tiempo de compilación, pero hay una cierta diferencia entre las que corresponden a datos y las que lo hacen a código, como una llamada. Como se explicó previamente, en el primer caso, se trata de una referencia desde la sección de código a una dirección en otra sección del mismo fichero objeto. En contraste, en el segundo caso, la referencia incluida en la sección de código corresponde a una dirección de esa misma sección, por lo que la distancia entre la instrucción que contiene la referencia y la referenciada no se verá afectada por el proceso de reubicación que se realiza en tiempo de montaje.

Cuestión 1

Usando la información obtenida por los tres mandatos (*nm*, *objdump* y *readelf*), explique de forma detallada qué tratamiento se le da a cada uno de los cinco símbolos, así como a las referencias a los mismos desde ambos módulos. Debe analizar razonadamente qué símbolos aparecen en la tabla de símbolos y cuáles no (*nm*), así como qué referencias se incluyen en la tabla de reubicaciones *(.rela.text*) y cuáles no, vinculándolo con el código ensamblador que corresponde a esas referencias.

Experimento de montaje

En esta fase se genera un ejecutable agrupando en regiones las secciones comunes de todos los ficheros objeto que forman parte de la aplicación, reubicando las direcciones asignadas a los símbolos para colocarlas en el espacio lógico del ejecutable, que para el procesador x86_64 abarca desde 0 a 2^{47}-1 (0x7fffffffffff; en este modelo de procesador son solo significativos los 48 bits de menor peso de la dirección lógica y en Linux se usa la primera mitad de ese espacio para direcciones de usuario y la otra mitad para direcciones de sistema). Por tanto, habrá que reubicar las referencias ya resueltas en tiempo de compilación para que correspondan a las nuevas direcciones asignadas a los símbolos. En esta etapa, además, hay que resolver las referencias a símbolos definidos en otros módulos. Nótese que para el ejercicio no se va a usar el mecanismo ASLR, especificando para ello la opción *-no-pie* a la hora de generar los ejecutables, por lo que el espacio lógico del ejecutable va a corresponder al espacio lógico del proceso en tiempo de ejecución, ya que consideramos que esta opción facilita desde un punto de vista didáctico la realización del ejercicio.

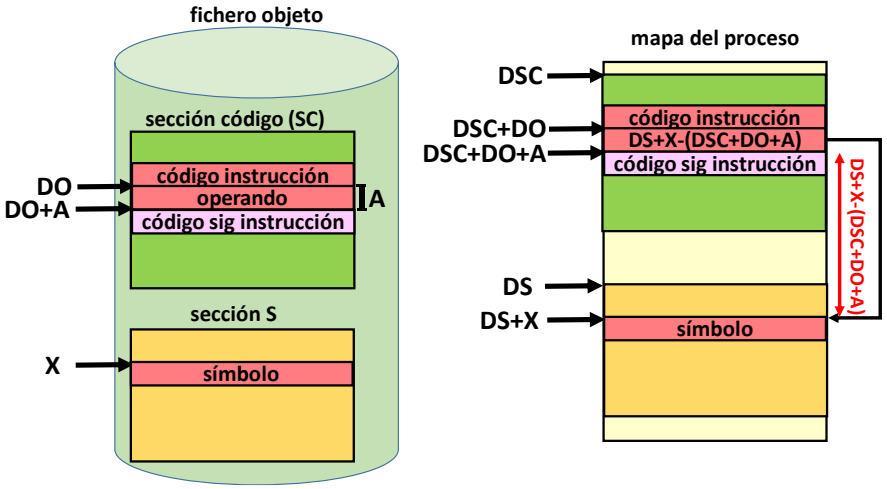
Para entender mejor este proceso, vamos a usar como ejemplo la sección que almacena los datos con valor inicial (*.data*). El montador agrupa todas las secciones *.data* de los ficheros objeto ubicándolas de forma contigua en el espacio lógico del ejecutable. Así, si a la sección *.data* del primer objeto se le ha asignado la dirección lógica *X* y ocupa *Y* bytes, esa misma sección del segundo objeto quedará situada en la dirección *X*+*Y* del espacio lógico, y así sucesivamente. Una variable global del segundo objeto que tuviera asignada en tiempo de compilación la dirección *Z* dentro de la sección *.data* de ese objeto tendrá asignada ahora la dirección *X*+*Y*+*Z* en el espacio lógico del ejecutable.

Siguiendo con el ejemplo, una vez reubicadas las direcciones asignadas a los símbolos, hay que pasar a reajustar las referencias resueltas en tiempo de compilación para que se correspondan con esas nuevas direcciones. Para ello, se usa la tabla de reubicaciones del objeto. Como se explicó en la sección anterior, una

entrada de esta tabla (*.rela.text*) que corresponda a una referencia resuelta en tiempo de compilación almacena la dirección del operando de esa instrucción (*DO*) dentro de esta sección de código, el identificador de la sección que contiene el símbolo referenciado (*S*; en el ejemplo, *.data*) y la dirección de ese símbolo dentro de esa sección (*X*), justo con un reajuste (*A*), que corresponde a la distancia entre el operando y la instrucción que sigue a la que hace la referencia, que como vimos suele valer -4. En tiempo de montaje, tanto la sección de código que contiene la instrucción que realiza la referencia como la sección que contiene el símbolo referenciado ya tienen asignadas direcciones en el espacio lógico del ejecutable (*DSC* y *DS*, respectivamente).

Recuerde que, dado que se usa un direccionamiento relativo al contador de programa, en el operando hay que guardar la distancia entre el símbolo y la instrucción que aparece después de la que realiza la referencia. Observe que si se resta de la dirección lógica asignada al símbolo (*DS+X*) la dirección lógica asignada al operando (*DSC+DO*), se obtendrá la distancia entre ambos. Sin embargo, como lo que se pretende calcular es la distancia entre el símbolo y la instrucción que sigue a la que hace la referencia, es necesario añadir el reajuste *A* al cálculo anterior, tal como se muestra en la siguiente figura:

(DS + X) - (DSC + DO + A), que equivale a *(DS + X - A) - (DSC + DO)*



Observe que para realizar el cálculo que aparece en la expresión equivalente de la derecha el montador obtiene de la entrada de reubicaciones del fichero objeto toda la información requerida:

- El identificador de la sección que contiene el símbolo referenciado (en este ejemplo, la sección *.data*). El montador conoce en qué dirección lógica (*DS*) ha quedado ubicada esa sección en el espacio lógico del ejecutable.
- La dirección del símbolo dentro de la sección y el reajuste requerido, que aparecen agrupados dentro de la entrada (*X - A*).
- La dirección del operando dentro de la sección de código del fichero objeto (*DO*; recuerde que es la primera columna que muestra el mandato *readelf -r*). Nótese que el montador conoce en qué dirección lógica (*DSC*) ha quedado ubicada esa sección de código en el espacio lógico del ejecutable.

A partir de esos valores, el montador realiza el cálculo (*(DS + X - A) - (DSC + DO)*) y actualiza el operando de la instrucción para que contenga el valor resultante.

Como último paso del proceso de montaje, hay que resolver las entradas de la tabla de reubicaciones de cada fichero objeto correspondientes a referencias no resueltas en tiempo de ejecución. En este caso, en la entrada

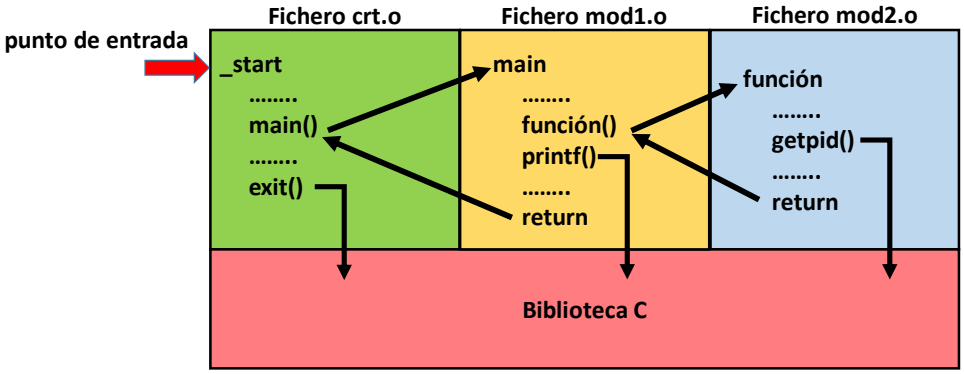
de la tabla aparece el identificador del símbolo pendiente de resolver, junto con el reajuste de -4 ya explicado. Recuerde que, como se explicó previamente, también aparecen como no resueltas las referencias a símbolos globales, aunque estén definidos en el mismo fichero objeto.

El proceso de resolución, en este caso, consiste en buscar qué dirección se le ha asignado a ese símbolo en el espacio lógico del ejecutable y calcular la distancia entre esa dirección (*D*) y la ubicación del operando en ese espacio lógico (*DSC + DO*), teniendo en cuenta el reajuste (*A*):

$$(D - A) - (DSC + DO)$$

A partir de esos valores, el montador realiza ese cálculo (*(D - A) - (DSC + DO)*) y actualiza el operando de la instrucción para que contenga el valor resultante.

Un aspecto que se ha obviado en esta explicación es que el montador tiene que añadir uno o más objetos adicionales con ciertas funcionalidades asociadas al lenguaje de programación correspondiente. En el caso del lenguaje C, el nombre habitual de estos ficheros comienza por *crt* (*C Run-Time*), y, además de otras funcionalidades, incluyen el punto de entrada del programa, que suele corresponder con una función denominada *_start*. La siguiente figura ilustra cómo sería el flujo de ejecución de un programa C formado por dos módulos y en el que se ha supuesto que solo se requiere un fichero objeto de tipo *crt*.



Cada uno de los ficheros objeto *crt* añadidos por el montador tiene las secciones habituales y participará, obviamente, en todo el proceso descrito en esta sección. Siguiendo con el ejemplo, las secciones *.data* de cada uno de los ficheros *crt* incluidos por el montador se agruparán con las de los objetos especificados por el usuario en el mandato de montaje.

Realice los siguientes pasos para llevar a cabo el experimento de montaje:

- Obtenga el código ensamblador de ese ejecutable y fíjese especialmente en el correspondiente a las referencias a los símbolos:

  objdump -S -Mintel main

  Tenga en cuenta que en ese código ensamblador está mezclado el de los objetos del programa con el de los módulos *crt* (se puede averiguar qué módulos *crt* usa el montador especificando la opción *-v* en el mandato *cc* que realiza el montaje; en la plataforma usada para preparar este ejercicio aparecen los siguientes ficheros *crt*: *crt1.o*, *crti.o*, *crtbegin.o*, *crtend.o* y *crtn.o*).

- Para afrontar la cuestión será necesario que averigüe en qué direcciones lógicas quedan ubicadas las secciones *.data* de los objetos. En principio, para ello, sería preciso conocer qué tamaño tiene esa sección en los ficheros *crt*. Esa información se puede deducir indirectamente si sabemos cuánto ocupa la sección *.data* de los objetos del programa y cuál es el tamaño total del agrupamiento de todas las

secciones de ese tipo. Para obtener estos datos, se puede usar el mandato *size* (también, se podría utilizar *readelf -S*):

size -Ax main.o | grep -w .data

size -Ax mod.o | grep -w .data

size -Ax main | grep -w .data

Cuestión 2

Realice los cálculos requeridos en tiempo de montaje por cada entrada de la tabla de reubicaciones de ambos módulos y verifique que los resultados corresponden a las referencias incluidas en el código ensamblador del programa.

Experimento de ejecución

En un sistema que usa un esquema de memoria virtual basado en paginación por demanda, el sistema operativo va asignando en tiempo de ejecución marcos de página a las páginas del proceso, usando tablas de páginas que establecen la correspondencia entre las direcciones lógicas que usa el proceso y las físicas asignadas al mismo, de manera que, en cada cambio de contexto, el sistema operativo informa a la MMU de qué tabla de páginas tiene que aplicar a partir de ese momento. El uso de esta etapa de traducción tanto para las direcciones de usuario como para las de sistema hace que no sea posible que un programa, o incluso el propio sistema operativo, pueda acceder directamente a una determinada dirección física de memoria.

En la gestión del espacio del proceso, el sistema operativo debe tener en cuenta las características de cada región del mapa del proceso: su ubicación, sus permisos, su origen, que puede ser un fichero o anónimo (por seguridad, se rellena con ceros) y si su carácter es compartido o privado, en cuyo caso se usa la técnica del COW para evitar que los cambios sobre una página de la región afecten al soporte original de la misma. El mandato *pmap PID* o el fichero */proc/PID/maps* proporcionan esta información de un proceso cuyo identificador es *PID*.

Activada la ejecución de un proceso e informada la MMU de qué tablas de páginas debe usar durante la ejecución del mismo, dado el uso de paginación por demanda para implementar el mecanismo de memoria virtual y de la técnica del COW para realizar por demanda las operaciones de duplicado requeridas por el manejo de regiones privadas y por la implementación de la operación *fork*, la asignación de memoria física al proceso se producirá únicamente en el tratamiento por parte del sistema operativo de dos excepciones:

- Fallo de página. El sistema operativo asigna un marco libre a la página solicitada rellenándolo con el contenido de la misma que puede proceder de un fichero o del *swap*, o requerir simplemente ser rellenado con ceros. Nótese que en Linux, si el fallo de página corresponde a una página de una región soportada en un fichero, para optimizar la operación del disco, se usa la técnica de *prepaginación* (*prepaging*, *read-ahead* o *prefetching*), que no solo trae a memoria la página referenciada sino también páginas adicionales que corresponden a las ubicadas en el fichero justo a continuación de la que causó el fallo. En la plataforma usada para preparar este ejercicio, se ha estimado empíricamente que en ese escenario se traen hasta 15 páginas adicionales.
- Fallo de protección (COW). El sistema operativo realiza un duplicado de esa página en un marco libre.

El objetivo de los experimentos de esta sección es reforzar todos estos conceptos de forma práctica, completando ese largo y tortuoso camino que transcurre desde que un programa referencia de forma simbólica a un determinado objeto hasta que esa referencia se convierte en un acceso a la dirección física de la celda de memoria asociada a ese objeto.

Para realizar los experimentos de esta sección, se proporcionan tres herramientas (directorio *herramientas* del material de apoyo) que **requieren ser ejecutadas como *superusuario***:

- *virt_to_phys*: imprime las direcciones físicas asociadas a direcciones lógicas de un proceso, siempre que la página correspondiente a cada una de ellas esté residente en memoria. Internamente, utiliza el fichero */proc/PID/pagemap*.
- *read_virt*: muestra el valor almacenado en una cierta dirección lógica de un proceso. Internamente, usa el fichero */proc/PID/mem*.
- *read_phys*: muestra el valor almacenado en una cierta dirección física. Esta herramienta usa un módulo del núcleo para llevar a cabo su labor. Aunque, en principio, podría parecer que se puede implementar accediendo al fichero */dev/mem*, en muchas plataformas este fichero está restringido de manera que solo se permite acceder a direcciones de memoria que estén asociadas a dispositivos de entrada/salida (MMIO).

Dada la volatilidad del API del núcleo, es posible que tenga problemas a la hora de usar la tercera herramienta en su equipo. Si se da esa circunstancia, no supone ningún impedimento ya que realmente esa herramienta es un poco redundante y proporciona, básicamente, la misma información que la segunda.

Como primer ejercicio de esta sección, vamos a usar el programa previo, pero utilizando una nueva versión (*main2.c*) del módulo que contiene la función principal del programa al que simplemente se le ha añadido un fragmento de código que imprime las direcciones de los dos objetos referenciados por el programa y que se queda esperando que se pulse alguna tecla para terminar.

A continuación, se detallan los pasos de este experimento:

1. Ejecute el programa *main2*. Nótese que la información que imprime este programa, y los posteriores, se puede copiar directamente como argumentos de la herramienta *virt_to_phys*, que es la más importante para estos experimentos.
2. Estando parado ese proceso, ejecute una segunda instancia del programa.
3. Termine la ejecución de ambas instancias.

Cuestión 3

Analice en el segundo paso, usando las herramientas proporcionadas (principalmente, *virt_to_phys*), las direcciones lógicas y físicas asignadas a los objetos referenciados. Compare usando *pmap* o el fichero *maps* de */proc/PID* los mapas de ambos procesos en ese segundo paso.

En los siguientes experimentos se pretende hacer un seguimiento del ciclo de vida de una página que, siguiendo la misma pauta que en las secciones previas, pertenezca a la región de datos con valor inicial. Para ello, se van a usar programas que definen un vector global con valor inicial de un tamaño relativamente grande y acceden a la parte final de ese vector. Para entender el uso de esa estrategia hay que tener en cuenta dos factores:

- En la primera página de la región de datos con valor inicial se incluyen objetos relacionados con la gestión de las bibliotecas dinámicas que son accedidos antes de que comience la ejecución de la función *main*. Por tanto, al comienzo de la ejecución de esa función, esa primera página ya está residente en memoria.
- Como se explicó previamente, Linux usa una estrategia de *prepaginación* en el tratamiento de los fallos de página vinculados con regiones que tienen soporte en un fichero. Por tanto, cuando se inicia la ejecución de la función *main*, no solo está residente esa primera página, sino un número significativo de páginas consecutivas de esa región.

Usando un vector relativamente grande y accediendo a la parte final del mismo nos aseguramos de que podemos hacer un seguimiento de todo el ciclo de vida de esa página.

Para el siguiente experimento vamos a usar un programa *main3* que va pasando por tres etapas: antes de acceder al vector, acceso de lectura al vector y modificación del vector.

Cuestión 4

Analice de forma razonada la evolución de las direcciones en las sucesivas etapas de ejecución del programa.

En este nuevo experimento, vamos a activar tres instancias del mismo programa (*main3*) según el siguiente plan:

1. Activo la primera instancia.
2. Activo la segunda instancia.
3. Pulso para que avance la primera instancia.
4. Pulso para que avance la segunda instancia.
5. Pulso nuevamente para que avance la primera instancia.
6. Pulso nuevamente para que avance la segunda instancia.
7. Activo la tercera instancia.
8. Pulso para que avance la tercera instancia.

Cuestión 5

Analice de forma razonada la evolución de las direcciones de los tres procesos en cada uno de los pasos de esa traza de ejecución.

En los siguientes experimentos, se crean procesos hijos y se analiza cómo influye en el ciclo de vida de esa página de la región de datos con valor inicial.

El siguiente experimento usa el programa *main4* de acuerdo con la siguiente traza:

1. Activo la ejecución del programa.
2. Pulso para que avance uno de los procesos.
3. Pulso para que avance el otro proceso.
4. Activo una segunda instancia del programa.

Cuestión 6

Analice de forma razonada la evolución de las direcciones de los procesos en cada uno de los pasos de esa traza de ejecución.

El siguiente experimento es igual que el anterior, pero usando el programa *main5*, cuyo único cambio respecto al anterior es que comienza actualizando el vector en vez de solo leyéndolo.

Cuestión 7

Compare de forma razonada los resultados de este experimento con los del anterior.

En los experimentos previos, hemos comprobado cómo los procesos que usan el mismo ejecutable, ya sean independientes o emparentados, comparten de forma real o mediante COW las regiones de su mapa vinculadas con el ejecutable. Esta compartición se extiende también a las bibliotecas dinámicas (esa es una de las ventajas del uso de bibliotecas dinámicas frente al de estáticas): procesos que usan diferentes ejecutables pueden compartir las bibliotecas dinámicas que utilicen en común.

En eso consiste este último ejercicio: active la ejecución de cualquiera de los programas previos (por ejemplo, *main2*) y verifique que comparte con el proceso *bash* (su PID es $$) la primera página de la región de código de la biblioteca *libc*.

Cuestión 8

Explique cómo ha realizado ese último ejercicio.

Plazo y modo de entrega

El plazo de entrega del trabajo es el 15 de junio de 2021.

La entrega se realiza en *triqui* ejecutando el mandato:

entrega.soa memoria.2021

Este mandato realizará la recolección del directorio *~/DATSI/SOA/memoria.2021* de los ficheros *autor.txt*, con los datos del alumno, y *memoria.pdf*, que debe incluir la solución del ejercicio.