16

Segmentación

Hasta ahora hemos estado guardando todo el espacio de direcciones de cada proceso en la memoria. Con los registros de base y límites, el SO puede reubicar fácilmente los procesos en diferentes partes de la memoria física. Sin embargo, es posible que hayas notado algo interesante acerca de estos espacios de direcciones: hay una gran cantidad de espacio "libre" justo en el medio, entre el stack y el heap.

Como podemos imaginar por la Figura 16.1, aunque el proceso no utiliza el espacio entre el stack y el heap, sigue ocupando memoria física cuando reubicamos todo el espacio de direcciones en algún lugar de la memoria física; por lo tanto, el enfoque simple de usar un par de registros de base y límites para virtualizar la memoria es un desperdicio. También hace que sea bastante difícil ejecutar un programa cuando todo el espacio de direcciones no cabe en la memoria;bpor lo tanto, la base y los límites no son tan flexibles como nos gustaría. Y por lo tanto:

LA CUESTIÓN: CÓMO SOPORTAR UN GRAN ESPACIO DE DIRECCIONES

¿Cómo soportamos un gran espacio de direcciones con (potencialmente) mucho espacio libre entre el stackby el heap? Hay que tener en cuenta que en nuestros ejemplos, con espacios de direcciones diminutos (simulados), el desperdicio no parece tan malo. Sin embargo, imaginemos un espacio de direcciones de 32 bits (4 GB de tamaño); un programa típico solo usará megabytes de memoria, pero aún así exigirá que todo el espacio de direcciones resida en la memoria.

16.1 Segmentación: Base/Límites generalizados

Para solucionar este problema, nació una idea y se llama **segmentación**. Es una idea bastante antigua, que se remonta al menos a principios de la década de los 60's [H61, G62]. La idea es simple: en lugar de tener solo un par de base y límites en nuestra MMU, ¿por qué no tener un par de base y límites por **segmento** lógico del espacio de direcciones? Un segmento es solo una porción contigua del espacio de direcciones de una longitud particular, y en nuestro espacio de direcciones canónico, tenemos tres segmentos lógicamente diferentes: código, stack y heap. Lo que la segmentación permite hacer al SO es colocar cada uno de esos segmentos en diferentes partes de la memoria física, y así evitar llenar la memoria física con espacio de direcciones virtuales no utilizado.

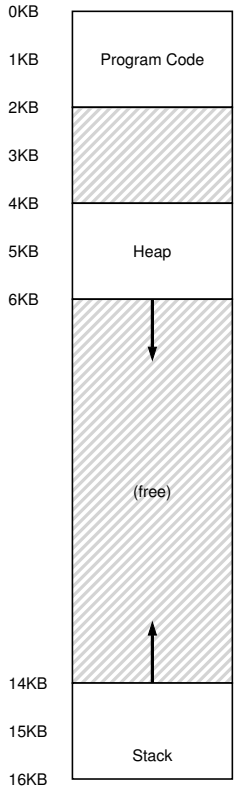


Figura 16.1: **Un Espacio de Direcciones (de nuevo)**

Veamos un ejemplo. Supongamos que queremos colocar el espa-

cio de direcciones de la Figura 16.1 en la memoria física. Con un par de base y límites por segmento, podemos colocar cada segmento *de forma independiente* en la memoria física. Para un ejemplo, consultar la Figura 16.2; allí se ve una memoria física de 64KB con esos tres segmentos (y 16KB reservados para el SO).

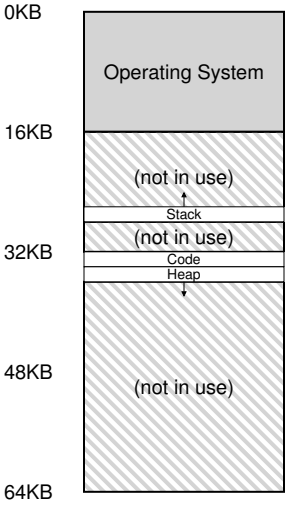


Figura 16.2: **Ubicando Segmentos en la Memoria Física**

Como se puede ver en el diagrama, solo a la memoria usada se le asigna espacio en la memoria física y, por lo tanto, se pueden acomodar grandes espacios de direcciones con grandes cantidades de espacio de direcciones sin usar (que a veces llamamos **espacios de direcciones dispersos**).

La estructura de hardware requerida en nuestra MMU para soportar la segmentación es justo lo que se esperaría: en este caso, un conjunto de tres pares de registros de base y límites. La Figura 16.3 a continuación muestra los valores de registro para el ejemplo anterior; cada registro de límites tiene el tamaño de un segmento.

Segmento	Base	Tamaño
Código	32K	2K
Heap	34K	3K
Stack	28K	2K

Figura 16.3: **Valores de los Registros de Segmentos**

Se puede ver en la figura que el segmento de código se coloca en la dirección física 32KB y tiene un tamaño de 2KB y el segmento de

APARTE: SEGMENTATION FAULT
El término **segmentation fault** o violación de segmento se produce a partir de un acceso a memoria a una dirección ilegal de una máquina segmentada. Cómicamente, el término persiste, incluso en máquinas que no soportan la segmentación en absoluto. O no tan cómicamente, si no podés darte cuenta por qué tu código sigue fallando.

stack se coloca en 34KB y tiene un tamaño de 3KB. El tamaño del segmento aquí es exactamente el mismo que el registro de límites introducido anteriormente; esto le dice al hardware exactamente cuántos bytes son válidos en este segmento (y por lo tanto, permite que el hardware determine cuándo un programa ha realizado un acceso ilegal fuera de esos límites).

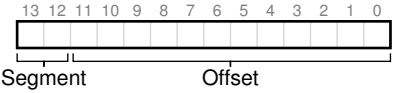
Hagamos una traducción de ejemplo, usando el espacio de direcciones de la Figura 16.1. Supongamos que se hace una referencia a la dirección virtual 100 (que está en el segmento de código, como se puede ver en la Figura 16.1, página 2). Cuando sucede la referencia (digamos, en un fetch de instrucción), el hardware sumará el valor base al *offset* en este segmento (100 en este caso) para llegar a la dirección física deseada: 100 + 32KB, o 32868. A continuación, comprobará que la dirección está dentro de los límites (100 es menos que 2KB), encontrará que lo está y emitirá la referencia a la dirección de memoria física 32868.

Ahora veamos una dirección en el heap, la dirección virtual 4200 (nuevamente, consultar la Figura 16.1). Si solo sumamos la dirección virtual 4200 a la base del heap (34KB), obtenemos una dirección física de 39016, que *no* es la dirección física correcta. Lo que primero debemos hacer es extraer el *offset* del heap, es decir, a qué byte(s) *de este segmento* se refiere la dirección. Debido a que el heap comienza en la dirección virtual 4KB (4096), el offset de 4200 es en realidad 4200 menos 4096, o 104. Luego tomamos este offset (104) y lo sumamos a la dirección física del registro base (34K) para obtener el resultado deseado: 4920.

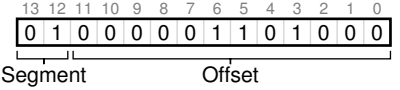
¿Qué pasa si intentamos hacer referencia a una dirección ilegal (es decir, una dirección virtual de 7KB o más), que está más allá del final del heap? Podemos imaginar lo que va a suceder: el hardware detecta que la dirección está fuera de los límites y hace un trap al SO, lo que probablemente lleve a terminar el proceso infractor. Y ahora conocemos el origen del famoso término que todos los programadores de C aprenden a temer: el **segmentation fault** o **violación de segmento**.

16.2 ¿A qué segmento nos referimos?

El hardware utiliza los registros de segmento durante la traducción. ¿Cómo conoce el offset en un segmento y a qué segmento se refiere una dirección? Un enfoque común, a veces denominado enfoque **explícito**, es dividir el espacio de direcciones en segmentos según los primeros bits de la dirección virtual; esta técnica se utilizó en el sistema VAX/VMS [LL82]. En nuestro ejemplo anterior, tenemos tres segmentos; por tanto, necesitamos dos bits para realizar nuestra tarea. Si usamos los dos bits superiores de nuestra dirección virtual de 14 bits para seleccionar el segmento, nuestra dirección virtual se ve así:



En nuestro ejemplo, entonces, si los dos bits superiores son 00, el hardware sabe que la dirección virtual está en el segmento de código y, por lo tanto, usa el par de base y límites del código para reubicar la dirección en la ubicación física correcta. Si los dos bits superiores son 01, el hardware sabe que la dirección está en el heap y, por lo tanto, usa la base y los límites del heap. Tomemos nuestra dirección virtual de stack de ejemplo de arriba (4200) y traduzcamos, solo para asegurarnos de que esto está claro. La dirección virtual 4200, en forma binaria, se puede ver aquí:



Como se puede ver en la imagen, los dos bits superiores (01) le dicen al hardware a qué *segmento* nos referimos. Los 12 bits inferiores son el *offset* en el segmento: 0000 0110 1000, o en hexadecimal 0x068, o 104 en decimal. Así, el hardware simplemente toma los primeros dos bits para determinar qué registro de segmento usar, y luego toma los siguientes 12 bits como offset en el segmento. Al sumar el registro base al offset, el hardware llega a la dirección física final. Hay que tener en cuenta que el offset también facilita la verificación de límites: simplemente podemos verificar si el desplazamiento es menor que los límites; si no, la dirección es ilegal. Por lo tanto, si la base y los límites fueran matrices (con una entrada por segmento), el hardware estaría haciendo algo como esto para obtener la dirección física deseada:

```
1 // obtener 2 bits superiores de direccion
2 // virtual de 14 bits
3 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
4 // ahora obtener offset
```

```
5  Offset = VirtualAddress & OFFSET_MASK
6  if (Offset >= Bounds[Segment])
7  RaiseException(PROTECTION_FAULT)
8  else
9  PhysAddr = Base[Segment] + Offset
10 Register = AccessMemory(PhysAddr)
```

En nuestro ejemplo de ejecución, podemos completar los valores de las constantes anteriores. Específicamente, `SEG_MASK` se establecería en `0x3000`, `SEG_SHIFT` en `12` y `OFFSET_MASK` en `0xFFF`.

Quizás notaste que cuando usamos los dos bits superiores, y solo tenemos tres segmentos (código, heap, stack), un segmento del espacio de direcciones no se usa. Para utilizar completamente el espacio de direcciones virtuales (y evitar un segmento no utilizado), algunos sistemas colocan el código en el mismo segmento que el heap y, por lo tanto, usan solo un bit para seleccionar qué segmento usar [LL82].

Otro problema con el uso de tantos bits superiores para seleccionar un segmento es que limita el uso del espacio de direcciones virtuales. Específicamente, cada segmento está limitado a un *tamaño máximo*, que en nuestro ejemplo es 4KB (usar los dos bits superiores para elegir segmentos implica que el espacio de direcciones de 16KB se divide en cuatro partes, o 4KB en este ejemplo). Si un programa en ejecución desea hacer crecer un segmento (digamos el heap o el stack) más allá de ese máximo, no va a tener suerte.

Hay otras formas para que el hardware determine en qué segmento se encuentra una dirección en particular. En el enfoque **implícito**, el hardware determina el segmento notando cómo se formó la dirección. Si, por ejemplo, la dirección se generó a partir del contador del programa (es decir, como un fetch de instrucción), entonces la dirección está dentro del segmento de código; si la dirección se basa en el puntero del stack o de la base, debe estar en el segmento del stack; cualquier otra dirección debe estar en el heap.

16.3 ¿Y el stack?

Hasta ahora, hemos omitido un componente importante del espacio de direcciones: el stack. El stack se ha reubicado en la dirección física 28KB en el diagrama anterior, pero con una diferencia crítica: crece hacia atrás (es decir, hacia direcciones más bajas). En la memoria física, "comienza" en 28KB[1] y vuelve a crecer a 26KB, lo que cor-

[1] Aunque decimos, por simplicidad, que el stack "comienza" en 28KB, este valor es en realidad el byte justo *debajo* de la ubicación de la región de crecimiento hacia atrás; el primer byte válido es en realidad 28KB menos 1. Por el contrario, las regiones de crecimiento positivo comienzan en la dirección del primer byte del segmento. Adoptamos este enfoque porque simplifica las matemáticas para calcular la dirección física: la dirección física es solo la base más el desplazamiento negativo.

responde a direcciones virtuales de 16KB a 14KB; la traducción debe realizarse de manera diferente.

Lo primero que necesitamos es un poco de soporte de hardware adicional. En lugar de solo valores base y límites, el hardware también necesita saber de qué manera crece el segmento (un bit, por ejemplo, que se establece en 1 cuando el segmento crece en la dirección positiva y 0 para la negativa). Nuestra vista actualizada de lo que monitorea el hardware se ve en la Figura 16.4:

Segmento	Base	Tamaño (max 4K)	¿Crece positivamente?
Código_{00}	32K	2K	1
Heap_{01}	34K	3K	1
Stack_{11}	28K	2K	0

Figura 16.4: **Registros de Segmentos (Con Soporte para el Crecimiento Negativo)**

El hardware, con el conocimiento de que los segmentos pueden crecer negativamente, ahora debe traducir dichas direcciones virtuales de manera ligeramente diferente. Tomemos un stack de direcciones virtuales de ejemplo y traduzcamos para comprender el proceso.

En este ejemplo, supongamos que deseamos acceder a la dirección virtual de 15KB, que debe asignarse a la dirección física de 27KB. Nuestra dirección virtual, en forma binaria, se ve así: 11 1100 0000 0000 (hexadecimal 0x3C00). El hardware usa los dos bits superiores (11) para designar el segmento, pero luego nos queda un offset de 3KB. Para obtener el offset negativo correcto, debemos restar el tamaño máximo del segmento de 3KB: en este ejemplo, un segmento puede ser de 4KB y, por lo tanto, el offset negativo correcto es 3KB menos 4KB, lo que equivale a -1KB. Simplemente agregamos el offset negativo (-1KB) a la base (28KB) para llegar a la dirección física correcta: 27KB. La verificación de límites se puede calcular asegurándose de que el valor absoluto del desplazamiento negativo sea menor o igual al tamaño actual del segmento (en este caso, 2KB).

16.4 Soporte para compartir

A medida que aumentaba el soporte para la segmentación, quienes diseñaban los sistemas pronto se dieron cuenta de que podían lograr nuevos tipos de eficiencia con un poco más de soporte de hardware. Específicamente, para ahorrar memoria, a veces es útil **compartir** ciertos segmentos de memoria entre espacios de direcciones. En particular, el **uso compartido de código** es común y todavía se usa en los sistemas de hoy.

Para admitir el intercambio, necesitamos un poco de soporte adicional del hardware, en forma de **bits de protección**. El soporte

básico agrega algunos bits por segmento, lo que indica si un programa puede leer o escribir un segmento, o quizás ejecutar código que se encuentra dentro del segmento. Al configurar un segmento de código como de solo lectura, el mismo código se puede compartir en varios procesos, sin preocuparse por dañar el aislamiento; mientras que cada proceso todavía piensa que está accediendo a su propia memoria privada, el SO está compartiendo en secreto memoria que no puede ser modificada por el proceso, y así se preserva la ilusión.

En la Figura 16.5 se muestra un ejemplo de la información adicional que debe tener el hardware (y el SO). Como se puede ver, el segmento de código está configurado para leer y ejecutar y, por lo tanto, el mismo segmento físico en la memoria podría mapearse en múltiples espacios de direcciones virtuales.

Segmento	Base	Tam. (max 4K)	¿Crece pos.?	protección
Código_{00}	32K	2K	1	Read-Execute
Heap_{01}	34K	3K	1	Read-Write
Stack_{11}	28K	2K	0	Read-Write

Figura 16.5: **Valores de los Registros de Segmentos (con Protección)**

Con los bits de protección, el algoritmo del hardware descrito anteriormente también debería cambiar. Además de verificar si una dirección virtual está dentro de los límites, también debería verificar si un acceso en particular está permitido. Si un proceso de usuario intenta escribir en un segmento de solo lectura o ejecutar desde un segmento no ejecutable, el hardware debe generar una excepción y, así, dejar que el SO se ocupe del proceso infractor.

16.5 Segmentación de grano fino vs. Segmentación de grano grueso

La mayoría de nuestros ejemplos hasta ahora se han centrado en sistemas con solo unos pocos segmentos (es decir, código, stack, heap); podemos pensar en esta segmentación como de **grano grueso**, ya que divide el espacio de direcciones en fragmentos relativamente grandes y gruesos. Sin embargo, algunos de los primeros sistemas (por ejemplo, Multics [CV65, DD68]) eran más flexibles y permitían que los espacios de direcciones constaran de una gran cantidad de segmentos más pequeños, lo que se conoce como segmentación de **grano fino**.

Soportar muchos segmentos requiere aún más soporte de hardware, con una tabla de segmentos de algún tipo almacenada en la memoria. Estas tablas de segmentos suelen admitir la creación de una gran cantidad de segmentos y, por lo tanto, permiten que un sistema utilice segmentos de formas más flexibles que las que hemos

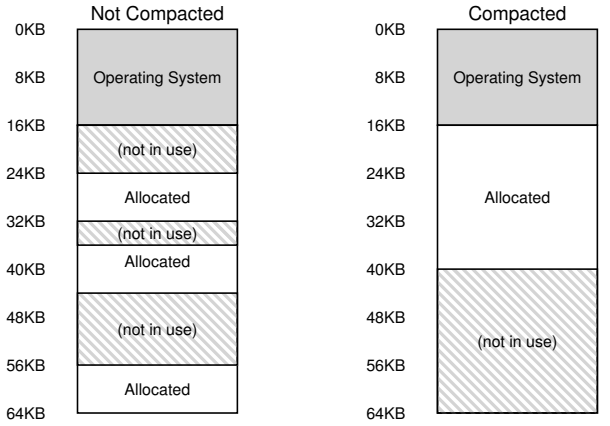


Figura 16.6: **Memoria no compactada y compactada**

analizado hasta ahora. Por ejemplo, las primeras máquinas como Burroughs B5000 tenían soporte para miles de segmentos y esperaban que un compilador dividiera el código y los datos en segmentos separados que el SO y el hardware admitirían [RK68]. La idea en ese momento era que al tener segmentos detallados, el SO podría aprender mejor qué segmentos están en uso y cuáles no y, por lo tanto, utilizar la memoria principal de manera más efectiva.

16.6 Soporte del SO

Ahora ya deberías tener una idea básica de cómo funciona la segmentación. Las partes del espacio de direcciones se reubican en la memoria física a medida que se ejecuta el sistema y, por lo tanto, se logra un gran ahorro de memoria física en relación con nuestro enfoque más simple con un solo par de base/límites para todo el espacio de direcciones. Específicamente, no es necesario asignar todo el espacio no utilizado entre el stack y el heap en la memoria física, lo que nos permite hacer entrar más espacios de direcciones en la memoria física y soportar un espacio de direcciones virtual grande y disperso por proceso.

Sin embargo, la segmentación plantea una serie de problemas nuevos para el SO. El primero es antiguo: ¿qué debería hacer el SO en un cambio de contexto? Deberías tener una buena idea a esta altura: los registros de segmento deben guardarse y restaurarse. Claramente, cada proceso tiene su propio espacio de direcciones virtuales, y el SO debe asegurarse de configurar estos registros correctamente antes de permitir que el proceso se ejecute nuevamente.

TIP:SI EXISTEN 1000 SOLUCIONES, NINGUNA ES GENIAL
El hecho de que existan tantos algoritmos distintos para intentar minimizar la fragmentación indica una verdad subyacente más fuerte: no existe la "mejor" forma de solucionar el problema. Así, nos conformamos con algo razonable y esperamos que sea lo suficientemente bueno. La única solución real (como veremos en capítulos futuros) es evitar el problema en su totalidad, al nunca asignar memoria en fragmentos de tamaño variable

El segundo es la interacción del SO cuando los segmentos crecen (o tal vez se reducen). Por ejemplo, un programa puede llamar a `malloc()` para asignar memoria a un objeto. En algunos casos, el heap existente podrá atender la solicitud y, por lo tanto, `malloc()` encontrará espacio libre para el objeto y devolverá un puntero al programa que la llamó. En otros, sin embargo, es posible que el segmento de heap tenga que crecer. En este caso, la biblioteca de asignación de memoria realizará una llamada al sistema para hacer crecer el heap (por ejemplo, la tradicional llamada al sistema UNIX `sbrk()`). Entonces, el SO (normalmente) proporcionará más espacio, actualizando el registro de tamaño de segmento al nuevo tamaño (más grande) e informando a la biblioteca del éxito; la biblioteca puede entonces asignar espacio para el nuevo objeto y regresar exitosamente al programa de llamada. Notar que el SO podría rechazar la solicitud si no hay más memoria física disponible o si decide que el proceso de llamada ya tiene demasiada.

El último problema, y quizás el más importante, es el manejo del espacio libre en la memoria física. Cuando se crea un nuevo espacio de direcciones, el SO debe poder encontrar espacio en la memoria física para sus segmentos. Anteriormente, asumíamos que cada espacio de direcciones tenía el mismo tamaño y, por lo tanto, la memoria física podría considerarse como un grupo de ranuras donde encajarían los procesos. Ahora, tenemos una cantidad de segmentos por proceso, y cada segmento puede ser de diferente tamaño.

El problema general que surge es que la memoria física se llena rápidamente de pequeños agujeros de espacio libre, lo que dificulta la asignación de nuevos segmentos o el crecimiento de los existentes. A este problema lo llamamos **fragmentación externa** [R69]; vea la Figura 16.6 (izquierda).

En el ejemplo, un proceso aparece y desea asignar un segmento de 20KB. En ese ejemplo, hay 24KB libres, pero no en un segmento contiguo (más bien, en tres fragmentos no contiguos). Por lo tanto, el SO no puede satisfacer la solicitud de 20KB. Pueden ocurrir problemas similares cuando llega una solicitud para hacer crecer un segmento; Si los siguientes bytes de espacio físico no están disponibles, el SO

tendrá que rechazar la solicitud, aunque puede haber bytes libres disponibles en otra parte de la memoria física.

Una solución a este problema sería **compactar** la memoria física reorganizando los segmentos existentes. Por ejemplo, el SO podría detener cualquier proceso que se esté ejecutando, copiar sus datos en una región contigua de memoria, cambiar los valores de registro de segmento para apuntar a las nuevas ubicaciones físicas y, por lo tanto, tener una gran cantidad de memoria libre con la que trabajar. Al hacerlo, el SO permite que la nueva solicitud de asignación se realice correctamente. Sin embargo, la compactación es cara, ya que la copia de segmentos consume mucha memoria y, por lo general, utiliza una buena cantidad de tiempo de procesador; consultar la Figura 16.6 (derecha) para ver un diagrama de la memoria física compactada. La compactación también hace que (irónicamente) las solicitudes para hacer crecer los segmentos existentes sean difíciles de atender y, por lo tanto, puede causar una mayor reorganización para acomodar dichas solicitudes.

En cambio, un enfoque más simple podría ser utilizar un algoritmo de administración *free-list* que intente mantener grandes extensiones de memoria disponibles para la asignación. Se han adoptado literalmente cientos de enfoques, incluidos algoritmos clásicos como el **best-fit** (que almacena una lista de espacios libres y devuelve el tamaño más cercano que satisface la asignación deseada), el **worst-fit**, el **first-fit** y esquemas más complejos como el **buddy algorithm** [K68]. Una excelente encuesta de Wilson et al. es un buen lugar para comenzar si se desea obtener más información sobre dichos algoritmos [W + 95], o se puede esperar hasta que cubramos algunos de los conceptos básicos en un capítulo posterior. Aunque, desafortunadamente, no importa cuán inteligente sea el algoritmo, la fragmentación externa seguirá existiendo; por eso, un buen algoritmo simplemente intenta minimizarlo.

16.7 Resumen

La segmentación resuelve una serie de problemas y nos ayuda a virtualizar la memoria efectivamente. Más allá de la relocalización dinámica, la segmentación puede soportar mejor los espacios de direcciones dispersos, al evitar el enorme desperdicio potencial de memoria entre los segmentos lógicos del espacio de direcciones. También es rápida, ya que realizar la segmentación aritmética es fácil y se adapta bien al hardware; los gastos generales de traducción son mínimos. También surge un beneficio adicional: el uso compartido de código. Si el código se coloca dentro de un segmento separado, dicho segmento podría potencialmente compartirse entre múltiples programas en ejecución.

Sin embargo, como aprendimos, la asignación de segmentos de

tamaño variable en la memoria genera algunos problemas que nos gustaría superar. El primero, como se discutió anteriormente, es la fragmentación externa. Debido a que los segmentos tienen un tamaño variable, la memoria libre se divide en trozos de tamaños extraños y, por lo tanto, puede resultar difícil satisfacer una solicitud de asignación de memoria. Se puede intentar utilizar algoritmos inteligentes [W+95] o periódicamente compactar la memoria, pero el problema es fundamental y difícil de evitar.

El segundo problema, y quizás el más importante, es que la segmentación aún no es lo suficientemente flexible para admitir nuestro espacio de direcciones disperso y generalizado. Por ejemplo, si tenemos un heap grande pero poco utilizado, todo en un segmento lógico, todo el heap debe residir en la memoria para poder acceder a él. En otras palabras, si nuestro modelo de cómo se usa el espacio de direcciones no coincide exactamente con cómo se diseñó la segmentación subyacente para respaldarlo, la segmentación no funciona muy bien. Por lo tanto, necesitamos encontrar algunas soluciones nuevas. ¿Listo/a para encontrarlas?

Referencias

[CV65] "Introduction and Overview of the Multics System" por F. J. Corbato, V. A. Vyssotsky. Fall Joint Computer Conference, 1965. *Uno de cinco artículos presentados en Multics en la Fall Joint Computer Conference; ojalá haber sido una mosca en la pared en esa habitación ese dia!.*

[DD68] "Virtual Memory, Processes, and Sharing in Multics" por Robert C. Daley y Jack B. Dennis. Communications of the ACM, Volumen 11:5, Mayo de 1968. *un artículo temprano sobre como realizar enlazamiento dinámico en Multics, lo cual era muy adelantado a su época. El enlazamiento dinámico finalmente logro reinsertarse en sistemas unos 20 años despues, ya que las grandes bibliotecas X-windows lo demandaban. ¡Algunos dicen que estas grandes bibliotecas X11 eran la venganza de MIT por la eliminación del apoyo al enlazamiento dinámico en versiones tempranas de UNIX!*

[G62] "Fact Segmentation" por M. N. Greenfield. Proceedings of the SJCC, Volumen 21, Mayo de 1962. *Otro artículo temprano sobre segmentación, tan temprano que no tiene referencias a otros trabajos.*

[H61] "Program Organization and Record Keeping for Dynamic Storage" por A. W. Holt. Communications of the ACM, Volumen 4:10, Octubre de 1961. *Un artículo increiblemente temprano y dificil de leer sobre segmentación y algunos de sus usos.*

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals" por Intel. 2009. Disponible: http://www.intel.com/products/processor/manuals. *Se puede intentar leer sobre segmentación aquí (Capítulo 3 en Volumen 3a); te hará doler la cabeza, al menos un poco.*

[K68] "The Art of Computer Programming: Volume I" por Donald Knuth. Addison-Wesley, 1968. *Knuth es famoso no solo por sus primeros libros sobre el Arte de Programación de Computadoras sino también por su sistema tipográfico TeX, el cual todavía es una potente herramienta de tipografía usada por profesionales a día de hoy, y de hecho compuso este mismo libro. Sus tomos sobre algoritmos son una gran referencia temprana para muchos de los algormos que subyacen a los sistemas de computación hoy.*

[L83] "Hints for Computer Systems Design" por Butler Lampson. ACM Operating Systems Review, 15:5, Octubre de 1983. *Un tesoro de sabios consejos sobre cómo construir sistemas. Dificil de leer en una sentada; es mejor tomarlo de a poco, como un buen vino, o un manual de referencias.*

[LL82] "Virtual Memory Management in the VAX/VMS Operating System" por Henry M. Levy, Peter H. Lipman. IEEE Computer, Volumen 15:3, Marzo de 1982. *Un sistema de manejo de memoria clásico, con mucho sentido común en su diseño. Lo estudiaremos con más detalle en un capítulo posterior.*

[RK68] "Dynamic Storage Allocation Systems" por B. Randell y C.J. Kuehner. Communications of the ACM, Volumen 11:5, Mayo de 1968. *Un buen vistazo a las diferencias entre paginación y segmentación, con algunas discusiones de varias maquinas.*

[R69] "A note on storage fragmentation and program segmentation" por Brian Randell. Communications of the ACM, Volumen 12:7, Julio de 1969. *Uno de los primeros artículos en discutir la fragmentación.*

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review" por Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. International Workshop on Memory Management, Escocia, UK, Septiembre de 1995. *Un gran artículo de encuesta sobre asignación de memoria.*

Tarea (Simulación)

Este programa permite ver cómo se realizan las traducciones de direcciones en un sistema con segmentación. Consultar el archivo README para más detalles.

Preguntas

1. Primero, usemos un pequeño espacio de direcciones para traducir algunas direcciones. A continuación, hay un conjunto simple de parámetros con algunas semillas aleatorias diferentes; traducir las direcciones:

   ```
   segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
   -L 20 -s 0
   segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
   -L 20 -s 1
   segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
   -L 20 -s 2
   ```

2. Ahora, veamos si entendemos este pequeño espacio de direcciones que hemos construido (usando los parámetros de la pregunta anterior). ¿Cuál es la dirección virtual legal más alta en el segmento 0? ¿Qué pasa con la dirección virtual legal más baja en el segmento 1? ¿Cuáles son las direcciones *ilegales* más bajas y más altas en todo este espacio de direcciones? Finalmente, ¿cómo se debería ejecutar `segmentation.py` con la bandera `-A` para probar que se está en lo cierto?

3. Supongamos que tenemos un pequeño espacio de direcciones de 16 bytes en una memoria física de 128 bytes. ¿Qué base y límites se debería establecer para que el simulador genere los siguientes resultados de traducción para el flujo de direcciones especificado: válido, válido, violación, ..., violación, válido, válido? Asumamos los siguientes parámetros:

   ```
   segmentation.py -a 16 -p 128
   -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
   --b0 ?  --l0 ?  --b1 ?  --l1 ?
   ```

4. Supongamos que queremos generar un problema donde aproximadamente el 90% de las direcciones virtuales generadas aleatoriamente son válidas (no *segmentation faults*). ¿Cómo se debería configurar el simulador? ¿Qué parámetros son importantes para obtener este resultado?

5. ¿Se puede ejecutar el simulador de manera que ninguna dirección virtual sea válida? ¿Cómo?