Virtualización de la Memoria

Ticiano Ian Morvan

Septiembre 2024

1 Abstracción

Al principio, solo un proceso corría acaparando todos los recursos de la computadora. Luego, se buscó que más procesos pudieran ejecutarse sobre una misma máquina, lo que llevó a diseñar técnicas de administración de recursos que permitieran este funcionamiento. Guardar todo el estado de un proceso en algún tipo de almacenamiento (desde discos hasta memoria física) fue una de las primeras ideas, pero rápidamente entró en un problema: guardar los contenidos de la memoria en este tipo de dispositivos es excesivamente lento. Por lo tanto, se presenta como una mejor idea el hecho de que el S.O. sea quien administre los procesos que corren y, por consiguiente, los recursos disponibles.

Es entonces que **abstraemos** la memoria física de forma que creamos, para cada proceso, un **espacio de redireccionamiento**. Éste contiene todo el estado de la memoria del proceso, el **código** del programa, su ***stack*** (variables locales, paso actual de la ejecución del programa, etc.) y su ***heap*** (memoria alojada dinámicamente por el usuario).

El resultado de esta abstracción es que el S.O. logra **virtualizar** la memoria física, provocando una suerte de ilusión a los procesos, que tienen la percepción de tener toda la memoria física disponible a su merced, cuando en realidad están alojados arbitrariamente por el S.O. en algún punto de la memoria física. Esta "ilusión" se conoce como ***isolation***.

La virtualización de memoria tiene como objetivo la **transparencia** (el programa no debe poder inferir que su memoria "fisica" es en realidad virtual), la **eficiencia** (en términos de espacio y tiempo, necesitando soporte del *hardware*) y la **protección** (un proceso no debe poder acceder ni modificar la memoria de otro).

2 API de Memoria

Tenemos dos tipos principales de memoria: el ***stack*** y el ***heap***. El *stack* es comúnmente llamada la "memoria automática", ya que es administrada implícitamente por el compilador, mientras que el *heap* es explícitamente administradas por el usuario.

```
1    #include <stdlib.h>
2    ...
3    int main(void) {
4      int x = 5;
5      int *y =
6        malloc (sizeof (int));
7      ...
8      free(y);
9    }
```

Listing 1: Uso de *stack* y *heap* en lenguaje C

En este ejemplo, `x` es una variable alojada estáticamente por el compilador, mientras que `y` es una variable alojada dinámicamente por el usuario, que debe ser liberada usando `free()`.

En el lenguaje C, no existe un *garbage collector* que libere de manera automática la memoria, en su lugar, el usuario es responsable de liberar la memoria que haya reservado, con el fin de evitar la aparición de *memory leaks*. Olvidarse de asignar memoria, asignar menos de la necesaria u olvidarse de inicializar la memoria reservada, son solo algunos de los problemas comunes que aparecen en la administración de memoria que propone este lenguaje.

Más allá de los posibles problemas de esta implementación, es notable la factibilidad de su uso, puesto que si bien es posible utilizar directamente *system calls* como `brk` para manejar la memoria de forma manual, esto podría incluso llevar a problemas de mayor escala, por lo que se recomienda altamente quedarse con `malloc()` y `free()`.

3 Traducción de direcciones

Para mantener el control sobre la memoria, impidiendo que un proceso monopolice o haga mal uso de los recursos, el SO debe trabajar directamente con el *hardware* para lograr un **manejo de memoria eficiente**, que le permita aislar a los procesos existentes como así también alojar nuevos o liberar los espacios ocupados anteriormente. Para hacerlo, entramos en el concepto de **traducción de direcciones basado en *hardware***, o simplemente **traducción de direcciones**, que permite al SO obtener la dirección **física** a la que estaba apuntando una dirección **virtual** que busca ser accedida por una instrucción.

Uno de los primeros métodos que se crearon fue el de ***base and bounds*** o también llamado ***dynamic reallocation*** que, en pocas palabras, compila cada programa arrancando desde la dirección **cero** pero, a través de un registro ***base*** que determina en que dirección de la memoria física está realmente alojado el programa y otro ***bounds*** el cual determina el tamaño máximo de memoria alojada para el mismo, por lo que si el programa busca acceder a una dirección **virtual**, primero sumamos la ***base***, de modo que

$$\text{direccion fisica} = \textit{base} + \text{direccion virtual}$$

Esto logra hacer **transparente** el manejo de la memoria para el proceso, ya que puede acceder a las direcciones sin mediar por ningún proceso manual de traducción, el *hardware* se encarga de hacerlo. El otro registro, ***bounds*** impide que el proceso pueda acceder a una dirección que está por fuera de su alcance, evitando que pueda leer o modificar información utilizada por otro proceso. Estos dos registros son parte estructural del *chip* de la CPU, a veces llamados ***memory management unit (MMU)***.

Un nivel de abstracción por encima, encontramos al sistema operativo, quien se encarga de administrar correctamente estas traducciones y los pasos intermedios para asegurar una fluida (¡y segura!) experiencia de usuario. El sistema operativo, para esta tarea, debe: asignar un nuevo proceso a un hueco libre de memoria, para ello llevando una "lista" de los espacios libres; liberar la memoria y las estructuras asociadas a un proceso terminado, salvar los registros de un proceso, sobre todo ***base*** y ***bounds***, así como la estructura del proceso en caso de ***context switch***; manejar excepciones para protegerse. Todo este proceso de traducción puede verse afectado por problemas como la **fragmentación interna** o la **segmentación**.

4 Manejo del espacio libre

Para combatir la **fragmentación externa**, es decir, cuando los espacios libres de la memoria se asignan en bloques de tamaño variable que en cierto punto imposibilitan la asignación de nuevos bloques, es que empezamos a hablar del **manejo de espacio libre**. Para ello, el sistema operativo lleva registro de aquellos espacios sin ocupar de la memoria en algún tipo de estructura, comúnmente llamada ***free list***.

4.1 Mecanismos de gestión

Conocemos en primer lugar dos mecanismos de bajo nivel para ayudarnos en el manejo del espacio libre. Por un lado, la **separación** o ***splitting*** permite que, si tenemos un espacio de n *bytes* libres y queremos asignar m *bytes* tales que $m < n$, en un primer lugar nos encontraríamos con **fragmentación interna**, puesto que el espacio asignado sería mayor al que realmente se necesita, desperdiciando ese espacio. A través de este mecanismo de separación, podemos **dividir** el bloque vacío y asignar solamente lo que se necesita, dejando el resto nuevamente libre. En segunda instancia, cuando liberemos esa memoria, veremos que, si estábamos llevando una lista de los espacios libres, ahora tendremos dos bloques separados. Esto puede llegar a ser problemático, puesto que si solicitamos cada vez espacios más pequeños de memoria, terminaremos fragmentándola en bloques que no puedan satisfacer ninguna asignación más que la mínima, lo cual es prácticamente inútil. Para ello, conocemos otro mecanismo: **fusión** o ***coalescing***. Éste permite **fusionar** aquellos espacios contiguos de memoria libre, logrando así recuperar secciones más grandes que sustenten las nuevas necesidades de asignación que se presentan. El último mecanismo que

podemos destacar es aquel que permita **incrementar el *heap*** (normalmente `sbrk` en sistemas UNIX) ya que, sin un mecanismo tal, podríamos quedarnos sin espacio asignable rápidamente y fallar inevitablemente.

Un punto a destacar de la gestión del espacio libre, es la adición de un *header* a cada espacio de memoria que se asigne, guardando en él información clave para su posterior liberación. Por ejemplo, agregar el tamaño ocupado por un bloque ayuda a determinar límites y facilita la recuperación de ese espacio de memoria.

4.2 Estrategias de gestión

Es entonces que empezamos a pensar, *¿cómo gestionamos el espacio libre?*. La primera estrategia ***best fit*** realiza una búsqueda sobre toda la lista de bloques libres que son más grandes que la memoria solicitada, devolviendo el menor de ellos. Esto busca reducir el espacio desperdiciado, pero tiene muchos problemas de rendimiento al tener que revisar toda la lista. De modo inverso, ***worst fit*** realiza nuevamente una búsqueda global sobre la lista, pero devuelve el espacio contiguo de memoria libre más grande. Si bien reduce la cantidad de pequeños bloques de memoria, está probado que rinde de forma inferior y lleva a una excesiva fragmentación. Por otro lado tenemos a ***first fit*** y ***next fit***. ***first fit*** devuelve el primer bloque de memoria que pueda soportar la asignación necesaria, lo que es más rápido pero a veces sobrecarga la parte inicial de la lista con pequeños objetos. Para combatir esto, ***next fit*** mantiene un puntero adicional que revisa la última posición vista.

Otras estrategias usadas son las **listas segregadas**, separando el manejo del espacio libre general del de algunos objetos particulares, haciendo más eficientes las asignaciones de estos tipos de objetos. Por otro lado, un ***buddy allocator*** divide los bloques de memoria en N fracciones. Si tomamos un ***binary buddy allocator***, tendríamos espacios de memoria divididos recursivamente en dos hasta poder encajar del mejor modo posible la asignación que necesitamos. Si bien esto crea muchos pequeños bloques y a veces puede desperdiciarse memoria al no poder asignar exactamente los *bytes* requeridos, es muy fácil de liberar puesto que encontrar el espacio contiguo consta solamente de cambiar un *bit*.