Gestión de Memoria: módulo `xalloc.c`

Objetivos

El objetivo del laboratorio es llegar a conocer el concepto de memoria dinámica de un proceso y la manera de gestionar este tipo de memoria. Esta gestión se circunscribe dentro del problema general de la asignación dinámica de espacio que aparece en otras situaciones dentro del campo de los sistemas operativos (así por ejemplo, en la multiprogramación con particiones variables o en la gestión del espacio de un sistema de archivos).

El laboratorio consistirá en diseñar y codificar, en lenguaje C y sobre el sistema operativo Linux, un módulo con rutinas que proporcionen mecanismos de gestión de memoria dinámica.

Descripción

Dentro del mapa de memoria de un proceso, existen tres segmentos: datos (*data*), pila (*stack*) y memoria dinámica (*heap*) que contienen información del proceso. Repasemos brevemente la correspondencia entre estos segmentos y los distintos tipos de datos que usan los programas.

En el **segmento de datos** se encuentran las variables globales y estáticas del programa. La declaración de esta clase de variables exige indicar el tamaño de las mismas, de manera que el compilador sepa cuánto espacio debe reservar para ellas.

En el **segmento de pila** se encuentran los registros de activación de las funciones a las que el proceso invoca, esto es, al conjunto formado por los argumentos de la llamada a la función, la dirección de retorno y las variables locales o automáticas. El tamaño de una variable local no ha de ser especificado en tiempo de compilación, basta con que sea conocido en tiempo de ejecución. Sin embargo, la información contenida en una variable local a una función se mantiene solamente durante la ejecución de dicha función.

La **memoria dinámica** es la que proporciona el soporte para programar estructuras de datos de tamaño variable (por ejemplo: árboles, listas, etc.) que persistan más allá del retorno de las funciones. La memoria dinámica se gestiona normalmente sobre espacio reservado en la parte superior del segmento de datos (conocido como *heap* o montículo) y por lo tanto será persistente. El sistema operativo proporciona las llamadas BRK y SBRK para modificar el tamaño de este segmento, pero no incluye servicios de gestión de memoria de propósito general. Sin embargo, la mayoría de las aplicaciones no usan directamente estas llamadas, sino que utilizan los servicios de la biblioteca **`malloc`** (`man malloc`) que proporciona un gestor de memoria dinámica de propósito general que permite que la aplicación en tiempo de ejecución reserve espacio para almacenar los objetos que vaya necesitando (función `malloc`) y libere dicho espacio cuando lo considere oportuno (función `free`). La biblioteca de gestión de memoria se encarga de guardar información de control que le permite determinar qué partes del *heap* están ocupadas en un determinado momento y cuáles están libres.

El laboratorio consistirá en construir un módulo de gestión de memoria dinámica, al que denominaremos `xmalloc`, con unas características muy similares a la biblioteca `malloc`.

Explicación breve del código de apoyo

Se tomará como punto de partida el código desarrollado y explicado en [Ker 88, Sección 8.7]. Dicho código se proporciona como material de apoyo. Es muy importante comprenderlo correctamente para poder desarrollar este laboratorio. En él están implementadas las funciones para reserva de espacio `xmalloc` y para liberación de espacio `xfree` (nuestras versiones de `malloc` y `free`).

Antes de plantear la labor que debe realizarse en este laboratorio, vamos a resaltar algunos de los aspectos más relevantes del código que se proporciona como material de apoyo.

- Se mantiene una lista circular de huecos ordenados por direcciones. La función `xmalloc` consulta esta lista al buscar un hueco para satisfacer una determinada solicitud de reserva. Para simplificar el código, esta lista tiene siempre al menos un componente: un falso hueco (base) de tamaño cero que se crea en la primera llamada a `xmalloc`.
- Cada hueco tiene asociada una cabecera (*Header*) que contiene un campo con la longitud del hueco medida en *tamaños de cabecera* (incluyendo la propia cabecera) y un puntero al siguiente hueco. Así, el espacio se asigna tomando como unidad el tamaño de la cabecera, redondeando por exceso el número de *bytes* pedidos en la llamada `xmalloc` a un número exacto de cabeceras. Se producirá por lo tanto fragmentación interna (parte del espacio asignado se desperdicia).
- Los bloques asignados no están en ninguna lista. Tiene asociada una cabecera igual que la de los huecos, pero donde el único campo significativo es el tamaño.
- La estrategia de asignación de espacio es *next-fit:* se utiliza, al igual que en *first-fit*, el primer hueco que se encuentre cuyo tamaño sea suficiente para satisfacer la petición pero, a diferencia de esta estrategia, se comienza cada vez la búsqueda desde donde termino la última. La variable global **`freep`** cumple esta función apuntando siempre al hueco que esta situado en la lista justo antes del hueco por el que comenzará la búsqueda. Esto es, la búsqueda de un hueco empieza siempre por el hueco siguiente al apuntado por **`freep`** (`freep->s.ptr`).
- Tanto el puntero que devuelve `xmalloc` como el que recibe `xfree` apuntan justo después de la cabecera del bloque, esto es, a la zona donde el usuario puede almacenar sus datos.
- Cuando `xmalloc` detecta que no puede satisfacer una petición, se invoca a SBRK (a través de la función interna `morecore`) para aumentar el tamaño de la zona de datos asignada al programa, y se añade de esta nueva zona a la lista de huecos, usando `xfree`.
- La rutina `xfree` se encarga de comprobar si el bloque que se libera genera un hueco que se compacta con otros huecos adyacentes.
- Cuando sólo se necesita usar parte de un hueco para satisfacer una petición, se utiliza la parte de direcciones más altas creándose un nuevo hueco con la parte de direcciones más bajas.
- Se resuelve el problema del alineamiento de los datos. En numerosas arquitecturas existen restricciones en la manera en que un determinado tipo de datos puede almacenarse en la memoria. Así por ejemplo, en muchas arquitecturas los enteros deben comenzar en una dirección que sea múltiplo de 4. Como el modulo del laboratorio no conoce el tipo de los datos que se guardarán en la zona pedida, deberá asegurarse que las direcciones que devuelve permitan almacenar a partir de ellas un valor del tipo de datos más restrictivo en esa arquitectura. Para ello se define un tipo `Align` y se fuerza que el tamaño de la cabecera sea múltiplo de este tipo definiendo ésta como una unión con un campo **`x`** de tipo **`Align`**.

Tarea - Rutinas de gestión de memoria

Partiendo del código comentado y MANTENIENDO ESTRICTAMENTE sus mismas características (política *next-fit,* idéntica cabecera, etc.*),* se pide incluir las siguientes características:

- Modificar el tamaño de la unidad de asignación de espacio. En lugar de asignar el espacio usando como unidad el tamaño de la cabecera, se usará el tamaño del tipo de datos más restrictivo en cuanto al alineamiento (para el Laboratorio se usará el tipo `Align` que está definido como `long`). El uso de una unidad de asignación más pequeña disminuirá el nivel de fragmentación interna. Algunos aspectos que hay que tener en cuenta al desarrollar estas modificaciones son:

  - El cambio en el tamaño de la unidad de asignación afecta el cálculo del número de unidades que se necesitan para satisfacer una petición así como el valor que se almacena en el campo `size` de la cabecera. En el código de apoyo se guarda en dicho campo el número de cabeceras que ocupa un

bloque incluyendo la propia cabecera. Con el cambio de la unidad de asignación, en su lugar se deberá almacenar el número de **`Aligns`** que ocupa en total el bloque (datos + cabecera).

- Otro aspecto que hay que revisar en el código original es el relacionado con la aritmética de los punteros. En el lenguaje C la expresión: `p + n`, siendo `p` un puntero y `n` un entero, dará un resultado diferente dependiendo del tipo de puntero. Así, si `p` es de tipo `(Header *)`, se le sumará al valor de `p` el tamaño de `n` cabeceras, mientras que si es de tipo `(Align *)` se le sumará `n` veces el tamaño del tipo `Align`. Para resolver este tipo de problemas puede ser necesario usar la operación de *cast* para transformar tipos de punteros. un ejemplo de este tipo de operación extraído del código original es el siguiente: `bp = (Header *)ap -1.`

- A pesar del cambio en el tamaño de la unidad de asignación, la función `morecore` debe modificarse para que siga pidiendo espacio, mediante SBRK, en zonas de 8K bytes (`NALLOC * sizeof(Header)`), siempre que el número de *bytes* requeridos no sea mayor que esta cantidad.

- En la implementación inicial pueden aparecer huecos de tamaño 1 cuando se asignan espacios en `xmalloc`, si en el espacio que sobra del hueco elegido, sólo cabe una cabecera. Estos huecos pueden ralentizar la búsqueda del hueco apropiado en posteriores llamadas a `xmalloc` ya que se tiene que consultar, aunque no sirvan para satisfacer ninguna petición. Se modificará el código para que la función `xmalloc` sólo genere un hueco sobrante si su tamaño es mayor o igual que el tamaño de la cabecera más el de una unidad de asignación. Esto es, si el espacio sobrante no sirve para satisfacer al menos una llamada `xmalloc` de 1 byte, no se genera el hueco y se deja que el bloque ocupe todo, apuntando en su cabecera el tamaño de todo el espacio. Esto es, se está transformando un problema de fragmentación externa en uno de fragmentación interna.

- Codificar la función `xrealloc` que cambia el tamaño de un bloque previamente reservado manteniendo su contenido inalterado hasta el mínimo de los tamaños nuevo y viejo. Para aclarar el comportamiento que debe tener esta función, analizaremos los distintos casos que se pueden presentarse.

  - Si la petición consiste en disminuir el tamaño, deberá generarse un hueco en la parte sobrante, siempre que su tamaño cumpla los requisitos del punto anterior. Un aspecto importante es que, aunque en principio la zona sobrante sea de un tamaño *inservible*, puede tener adyacente un hueco con el que se puede compactar, debiéndose producir en este caso la generación y compactación del hueco. En resumen, solo no se genera un hueco con la parte sobrante si su tamaño es insuficiente y además no es adyacente a un hueco. En el caso de que se genere un hueco, la variable **`freep`**, deberá quedar apuntando al hueco anterior al generado, osea, siguiendo el mismo criterio que la la función `xfree`.

  - Si la petición consiste en aumentar el tamaño y existe un hueco adyacente a la parte final del bloque con un tamaño suficientemente grande, no será necesaria la reubicación (o sea, mover el contenido del bloque a otra zona lo cual es una operación costosa). Sólo será necesario, si se cumple los requisitos de tamaño mínimo, generar un nuevo hueco con la parte sobrante del hueco adyacente usado y ajustar el nuevo tamaño del bloque. En este caso, la variable **`freep`** deberá quedar apuntando al hueco anterior al usado en la expansión.

  - En caso de que se trate de una petición de aumentar el tamaño y, o bien no hay un hueco adyacente en la parte final o bien el tamaño del mismo no es suficiente, se comprobará la existencia de un hueco adyacente a la parte inicial. Si el espacio que ocupa el bloque más el espacio del hueco adyacente por la parte final, si existe, más el espacio del hueco adyacente por la parte inicial es suficiente para satisfacer la petición, se usará el espacio correspondiente a las tres zonas (o dos, si no hay adyacencia por la parte final) para reubicar el bloque. O sea se considerará el intervalo formado por esas tres (o dos) zonas como un hueco y se reservará en dicho hueco el espacio necesario para el

nuevo tamaño del bloque (siguiendo los mismos criterios que `xmalloc` en la forma de reservar el espacio, o sea, usar la zona de direcciones más altas, y de actualizar **`freep`**), pasando a continuación copiar el contenido original al nuevo destino.

- En el caso de que el tamaño del bloque más el del espacio adyacente por ambos lados, si existe, no sean suficientes para satisfacer el aumento de tamaño solicitado, se buscará espacio para el nuevo tamaño del bloque usando el orden *next-fit*, se copiará el contenido al nuevo destino y, por ultimo, se liberará la zona original ocupada por el bloque. La variable **`freep`**, por lo tanto, quedará con el valor correspondiente a la liberación del espacio original.

La descripción de la función sería:

```
void * xrealloc(void * ptr,size_t size);
```

Cambia el tamaño del objeto al que apunta `ptr` a `size`. El contenido permanecerá sin alteraciones hasta el mínimo de los tamaños nuevo y viejo. Devuelve un puntero al nuevo espacio, o `ptr` si no se reubicó. En caso de error devuelve `NULL`

Programas a implementar

1 `xalloc`: Código fuente y ejecutable del módulo de gestión de memoria dinámica. En él se encontrarán todas las rutinas necesarias que tendrá que modificar según lo descrito anteriormente. Junto a la bibliografía se proporciona como material de apoyo el código de alguna de las funciones que usted tendrá que modificar y/o implementar. Entre las funciones que se encuentran son `xmalloc`, `xfree`, `xrealloc`, etc.

2 Durante el laboratorio se le proporcionará un código que permita verificar que sus rutinas están correctamente modificadas y/o implementadas. Las salidas de dichos ejecutables también se presentarán al final del laboratorio. Usted previamente podrá ir probando sus rutinas. pero sólo consideraremos válidas las salidas que nos entregue como respuestas el día del laboratorio.

Material de Apoyo

Makefile Archivo fuente para la herramienta `make`. No debe ser modificado. Con él se consigue la recompilación automática sólo de los ficheros fuente que se modifiquen.

xalloc.h Archivo fuente de definiciones de C que define las constantes y los prototipos de las funciones a implementar. No debe ser modificado.

xalloc.c Archivo fuente de C correspondiente al módulo a implementar. Inicialmente incluye la implementación descrita en [Ker 88, sección 8.7]. Este fichero es el que debe de modificar.

xrun.c Archivo fuente de C correspondiente a probar los módulos contenidos en *xalloc.c*

Bibliografia

[Ker 88] El Lenguaje de programación C, Brian W. Kernighan, Dennis M. Ritchie

Segunda Edición, Prentice-Hall, 1988.

[Roc 85] Advanced UNIX Programming, M.J. Rochkind

Prentice-Hall, 1985.

[Ste 92] Advanced Programming in the UNIX Environment, W. Richard

Stevens Addison-Wesley, 1992.