Trabajo práctico especial anexo

Arquitectura de las computadoras

Primer Cuatrimestre 2015

Autores:

Braulio Sespede - 51074
Mauricio Minestrelli - 52015

1. Objetivo

Implementar sobre el trabajo práctico especial las funciones *malloc* y *free* en el userland. Ambas implementaciones deben respetar la funcionalidad de las mismas en la librería estándar.

Además se deberá implementar la syscall sbrk la cual dará espacio físico para que malloc y free operen sobre el heap.

2. Kernel space

2.1. Heap

En nuestra implementación se seteo inicialmente el heap al final del binario del kernel con un tamaño máximo de 5MB (no por una limitación de memoria física, sino una limitación que permita a otros procesos hacer uso eficiente de los recursos del sistema). Este límite está definido como una constante en la MMU en el kernel, y podría cambiarse si fuera necesario.

Luego, por complicaciones con el tamaño del binario tuvimos que mover el heap a otra zona de memoria, alejada del mismo.

User space podrá pedirle al kernel que le asigne espacio de memoria manipulable para el heap mediante la system call sbrk. Si user space intentase utilizar dicha memoria sin haber hecho sbrk obtendrá una excepción ya que no tendría permiso para leer/escribir dicha memoria. A su vez, fué necesario implementar una system call adicional la cual retorna la dirección de comienzo del heap. Estas serán explicadas en las secciones correspondientes.

2.2. System call sbrk

En nuestra implementación la system call srbk aumenta la memoria disponible para que userland utilice como heap en 8KB (definido por una constante en la MMU). En el caso de que se exceda el tamaño máximo al intentar la operación retorna ENOMEM, caso contrario retorna el puntero al final de la zona disponible para uso del heap.

La zona de comienzo del heap es asignada a la MMU pasandole un puntero a final del binario del kernel en la inicialización de la misma.

En la implementación original de POSIX se permite pasarle un parametro a sbrk indicando la cantidad de bytes que quiere asignar, permitiéndole un uso más eficiente de los recursos. Además se puede obtener la posición actual pasándole como parámetro un 0, o incluso liberar memoria pasandole un número negativo.

2.3. System call getHeapBaseAddress

Encontramos que en muchas implementaciones se aprovechaba el parámetro de sbrk para obtener el comienzo de la misma pasandole un 0 como parámetro al sbrk. Lo cual permite obtener el comienzo del heap o la posición actual sin ningún tipo de inconveniente.

Dado que nuestra implementación de sbrk pedía implementar sbrk sin ningún tipo de parámetro, no había manera de obtener el comienzo de la zona reservada dado que sbrk siempre retorna un puntero al final de la misma (y siempre reserva 8KB en cada llamada). Se podría haber implementado que sbrk retorna un puntero al comienzo de la zona reservada pero esto implicaría que user space conociese el tamaño del incremento de sbrk, razón por lo cual no se hizo.

En cambio optamos por implementar una syscall adicional que retorna un puntero a la dirección base del heap.

3. User space

3.1. Consideraciones en la implementación

Junto con la definición de la implementación del malloc se debió definir la estructura de los bloques de memoria, las cuales son escritas sobre el heap y preceden al dato mismo. Esta estructura describe el tamaño del dato (en bytes), si se encuentra libre o reservado y el bloque siguiente en memoria (un puntero al mismo). Posteriormente se agregó también un puntero al bloque anterior para poder optimizar la liberación de memoria uniendo bloques libres hacia atrás y hacia adelante.

```
struct block{
       int size;
       type_block next;
       type_block prev;
       int free;
};
```

Es importante aclarar que no realizamos alineamiento a 8 bytes de los datos ni de la estructura. Si bien esto hubiese mejorado significativamente la performance nos pareció que se desperdicia demasiada memoria y está fuera del scope de nuestra implementación.

Nota: De ahora en más cuando nos refiramos al bloque nos estaremos refiriendo tanto a la estructura como al dato que describe.

3.2. Malloc

```
void * malloc(int size);
```

La función malloc devuelve puntero a una zona de memoria de lectura/escritura en el heap determinado por la cantidad de bytes requeridos por el usuario. En el caso de no disponer de más memoria para el heap para reservar la función devuelve NULL.

3.2.1. Implementación

El funcionamiento del malloc consiste en primer lugar en chequear si la variable global que indica el comienzo del heap ha sido inicializado. Si lo fue llama a *findBlock*, función que busca el primer bloque libre donde entre un nuevo bloque (el pedido por malloc). Al encontrar procede a dividirlo en dos, una parte será el bloque pedido y la otra parte quedará libre para un futuro malloc.

En caso de no encontrar un bloque libre procede a llamar a *expandHeap*, el cual hace tantas llamadas a *sc_sbrk* como sean necesarias para poder insertar el nuevo bloque en el heap.

Si la variable global núnca ha sido inicializada se le asigna el puntero retornado por la syscall *sc_getBaseHeapAddress* (recordar que no se puede hacer sbrk(0)) y se procede a expandir como se ha dicho previamente teniendo en cuenta que es el primer bloque del heap.

Si en cualquiera de las instancias la expansión hubiese fallado por falta de memoria se retorna *NULL*.

3.2.2. Funciones auxiliares

```
type_block findBlock(type_block *lastBlock, int size);
type_block splitBlock(type_block b, int size);
type_block expandHeap(type_block lastBlock, int size);
```

3.2.3. findBlock

Itera por la estructura buscando un bloque de tamaño mayor al tamaño del bloque que se quiere reservar y lo devuelve, caso contrario devuelve *NULL*. A su vez modifica el puntero al último visitado de manera que *expandHeap* puede aprovecharlo y no tenga que volver a recorrer la estructura.

3.2.4. splitBlock

Esta función divide el bloque recibido por parámetro en dos partes, de acuerdo al tamaño indicado. La segunda parte del bloque original se marca como libre y la primera es utilizada por malloc para aprovechar el bloque libre y es del tamaño pasado por parámetro.

3.2.5.expandHeap

Esta función hace tantos llamados a *sbrk* como sean necesarios para poder colocar en el heap un bloque de tamaño pasado por argumento. En caso de no poder asignarlo por limitación de memoria del *sbrk* retorna *NULL*, valor el cual es luego retornado por malloc.

Una vez que se está seguro que habrá espacio de lectoescritura en el heap se procede a escribir el bloque de tamaño pasado por argumento en el heap en la posición donde corresponda (se utiliza el parámetro *lastBlock* como referencia para saber donde hay que colocar el bloque nuevo).

3.3. Free

```
void free(void * address);
```

La función free libera el bloque de memoria reservado en el heap descrito por el puntero al dato pasado por parámetro. El principal problema que trae aparejado las operaciones malloc-free es la fragmentación externa que producen. Esto sucede ya que al subdividir bloques muchas veces quedan bloques de tamaño extremadamente chico, lo cual dificulta su futuro uso. Como consecuencia quedan “agujeros de memoria” en el heap que están siendo desaprovechados.

Es por esta problemática que se decidió implementar la unión de los bloques contiguos al momento de la liberación de un bloque para así reducir la fragmentación, al menos parcialmente. Es para esto que se utiliza el puntero al bloque previo antes mencionado.

Si bien se podría haber eliminado la fragmentación acomodando los bloques de manera contigua luego de un free, es un operación de orden N y puede llegar a ser muy costosa para una operación como el free, que debe ser ágil.

3.3.1 Implementación

La lógica del free consiste en marcar como libre el bloque asociado a la dirección de memoria que se pasa por parámetro (o cambiando el puntero del ante último a *NULL* en caso de ser el último bloque el liberado). De ser inválida la dirección free no hace nada (en un sistema verdaderamente protegido se lanzaría una excepción de segmentación por falta de permiso).

Una vez que se libera el bloque asociado a la dirección, se intenta unir este bloque con el anterior y/o el siguiente en el caso de que ambos estén libres mediante la función *mergeFreeBlocks* . En el caso en el que el bloque resultante esté al final del heap se reduce el tamaño del mismo y esto constituye también un mecanismo para evitar la fragmentación.

3.3.2. Funciones auxiliares

```
int validAddress(void * address);
type_block mergeFreeBlocks(type_block prev, type_block blockToFree);
type_block getBlockFromAddress(void * address);
```

3.3.3. validAddress

Devuelve si la dirección pasada por parámetro está dentro de los límites del heap.

3.3.4. mergeFreeBlocks

Esta función une dos bloques vacíos, sumándole al primero el tamaño del segundo y haciendo apuntar al próximo del primero al próximo del segundo. De ser posible (no nulo), se setea el previo del siguiente al segundo bloque como el primer bloque.

3.3.5. getBlockFromAddress

Devuelve la dirección de comienzo de bloque a partir de la dirección de comienzo de la data del mismo. Solo es llamado una vez verificado que la dirección es válida o podría funcionar incorrectamente.

3.4 Funciones de testeo

```
int exec_print_heap(int argc, char **argv)
int exec_free(int argc, char **argv)
int exec_malloc(int argc, char **argv)
int exec_string_malloc(int argc, char **argv){
```

3.4.1. malloc

Esta función reserva tantos bytes como se pasen por argumento y luego muestra en consola la dirección donde se reservó la memoria para el dato. Al recibir un argumento e interpretarlo como *int*, si el argumento es mayor al *int* máximo la cantidad reservada depende del número resultante del overflow (si es menor a cero se hace un *malloc* de 0 bytes).

3.4.2. smalloc

Funcionamiento idéntico a malloc pero recibe como argumentos tantos strings como se deseen, los cuales son insertados en el heap, reservando la cantidad adecuada de memoria para cada uno de ellos.

3.4.3. pheap

Esta función imprime en pantalla el estado del heap, mostrando los bloques que lo componen.

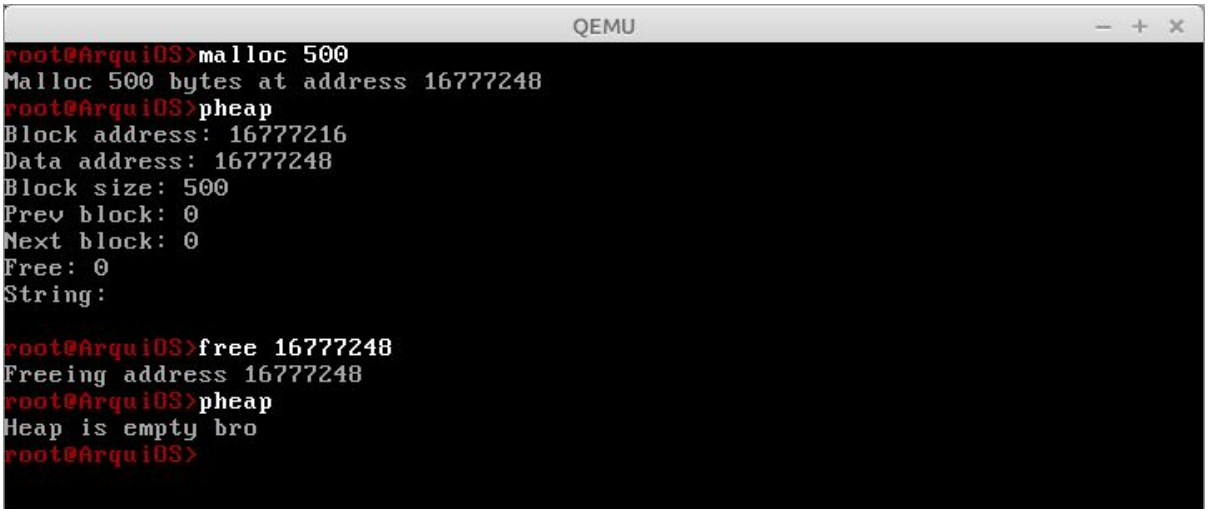
3.4.4 free

Esta función libera la dirección pasada como argumento (en formato decimal).

4. Capturas de pantalla

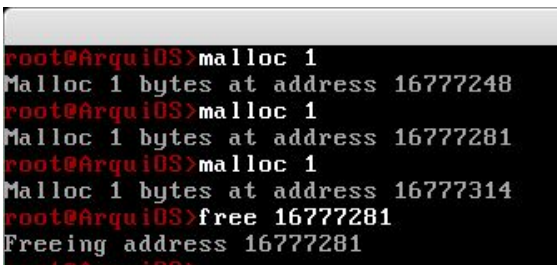
4.1. Funcionamiento del malloc, vaciado del heap.

En este ejemplo el heap está vacío, luego se hace un malloc de 500 bytes y la data se ubica en la dirección 16777248, luego se libera esta dirección y el heap queda libre nuevamente.



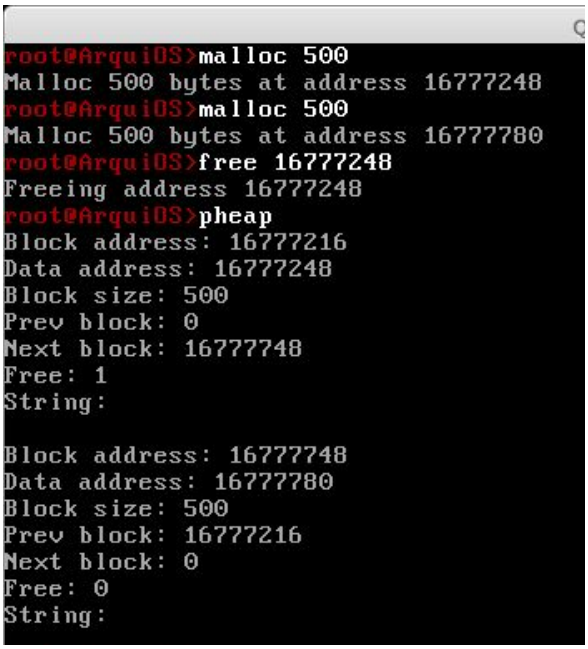
4.2. Funcionamiento del free

Se reserva memoria tres veces y luego se libera el bloque del medio en este caso no hay un merge posible.



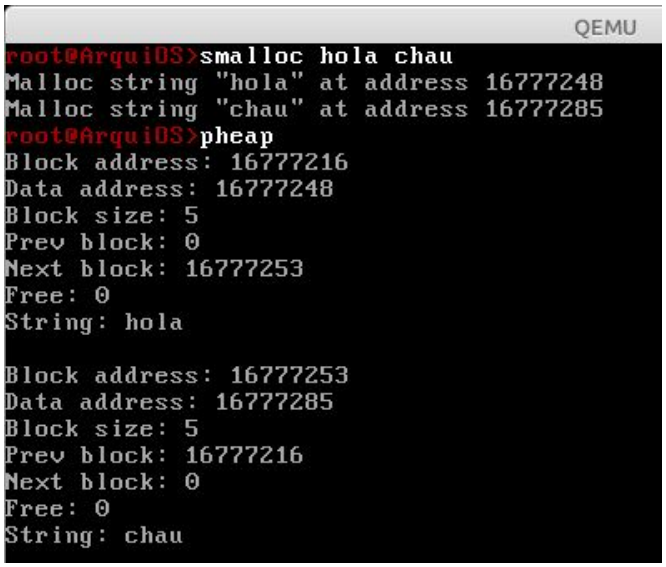
4.3. Split de bloques

Se reservan dos bloques de 500 bytes y luego se libera el primero, quedando al principio un bloque libre. Luego se hace un malloc de 250 bytes que se ubica en el bloque libre dividiéndolo en dos, un bloque que se ajusta a su tamaño y otro del tamaño restante.



4.4 Funcionamiento del smalloc

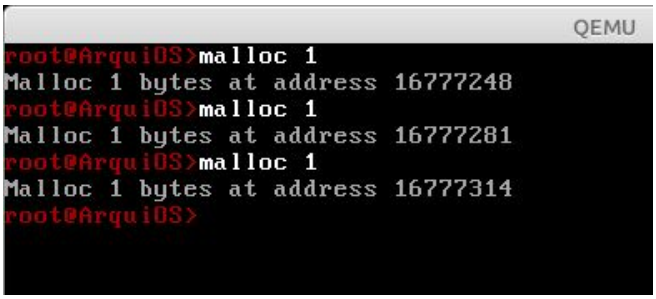
Se reserva memoria para los strings “hola” y “chau” pasados como argumentos, luego se muestra el estado del heap.

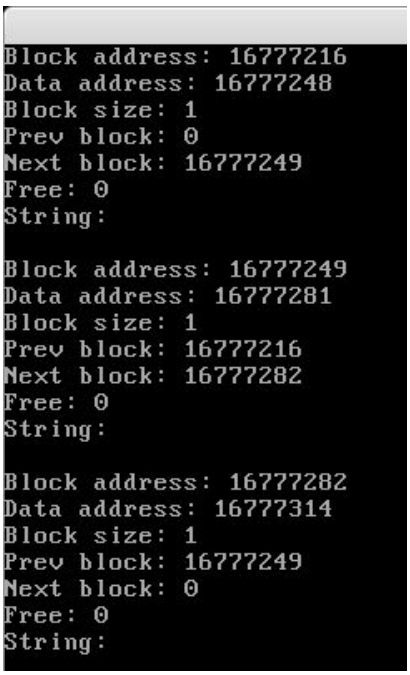


.

4.5 Funcionamiento del merge de bloques libres.

Se reservan tres bloques de un byte y luego se liberan los dos primeros. Al liberarse los dos primeros se *mergean* debido a que ambos están libres y contiguos.





5. Referencias

- A Malloc Tutorial∗ Marwan Burelle† Laboratoire Systeme et Sécurité de l'EPITA (LSE) ´ February 16, 2009 (http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf).
- A Quick Tutorial on Implementing and Debugging Malloc, Free, Calloc, and Realloc(http://danluu.com/malloc-tutorial/).
- A memory allocator by Doug Lea (http://g.oswego.edu/dl/html/malloc.html).