P4. GESTIÓN DE MEMORIA DINÁMICA PARA PROCESOS

Sistemas Operativos 1
Mayo 2023

1 LA LLAMADA AL SISTEMA SBRK

El objetivo de esta práctica es realizar una implementación propia de la gestión de memoria dinámica en un proceso a través de la implementación de las funciones de la librería estándar `malloc` y `free`. Estas dos funciones no son llamadas a sistema, sino que son llamadas que forman parte de la librería de usuario. Internamente utilizan llamadas a sistema.

¡Atención! Esta práctica no funcionará en sistemas Mac ni en máquinas virtuales. La firma de la función `malloc` (memory allocation) de C es la siguiente: `void *malloc(size_t size)`. Como parámetro de entrada recibe un número de bytes a reservar y devuelve un apuntador al bloques de datos que se ha reservado. Una forma de implementar el `malloc` es utilizando la llamada a sistema `sbrk`, una función que permite manipular el espacio de heap del proceso.

La función `sbrk` puede interpretarse intuitivamente como una función que permite aumentar o disminuir la cantidad de agua (memoria dinámica) asociada al pantano (proceso). La llamada `srbk(0)` devuelve el nivel del agua actual del pantano (un apuntador al nivel actual del heap). Si especificamos cualquier otra cantidad como parámetro podemos aumentar o disminuir el nivel del agua del pantano, el heap se incrementa o disminuye en este valor y la función devuelve un apuntador al valor antiguo antes de realizar la llamada.

Así, por ejemplo, la llamada `sbrk(1000)` aumenta en 1000 bytes el heap y devuelve un apuntador a el inicio de estos 1000 bytes de forma que se puedan utilizar los 1000 bytes por la aplicación. Observar, en cambio, que si se hace la llamada `sbrk(-1000)` se disminuye en 1000 bytes el espacio de memoria asociado en la heap, el valor devuelto es un apuntador en el nivel de heap antes de realizar la llamada. La función `sbrk(size)` devuelve un -1 en caso de que no se haya podido realizar la operación deseada.

2 UNA VERSIÓN DUMMY DE MALLOC I FREE

Se propone a continuación una implementación bien sencilla de las funciones `malloc` i `free`. El fichero asociado es llama `malloc_dummy.c`

```c
void *malloc(size_t mida) {
  void *p = sbrk(0);

  fprintf(stderr, "Malloc\n");

  if (mida <= 0)
    return NULL;

  if (sbrk(mida) == (void*) -1)
    return NULL; // sbrk failed.

  return p;
}

void free(void *p)
{
  fprintf(stderr, "Free\n");
}
```

Observar la implementación de este `malloc`. Esta implementación del `malloc` tiene el inconveniente que no podemos hacer un `free` de la memoria ocupada cuando no la necesitemos

dado que la función `sbrk` sólo nos permite aumentar o disminuir el nivel del *heap*, pero no permite liberar "un trozo" del medio del *heap*. Esto ocurre a menudo en las aplicaciones dado que iremos reservando y liberando memoria dinámica a lo largo de la vida del proceso.

Con la implementación del archivo `malloc_dummy.c` la memoria se acabaría llenando rápidamente con aplicaciones como el *firefox* dado que sólo vamos aumentando el nivel de *heap* (el agua del pantano), pero podemos probar si el código funciona con aplicaciones sencillas. Aquí se muestra un pequeño código en C que probaremos con la implementación del `malloc` que hemos hecho, código `exemple.c`

```c
int main()
{
    int i;
    int *p;

    p = malloc(10 * sizeof(int));

    for(i = 0; i < 10; i++)
        p[i] = i;

    for(i = 0; i < 10; i++)
        printf("%d\n", p[i]);

    free(p);

    return 0;
}
```

Esta función llama a la función `malloc`. El objetivo es generar un ejecutable de forma que se utilice la función `malloc` que acabamos de definir en vez de la función `malloc` de la librería estándar. Por eso es necesario ejecutar las siguientes instrucciones en un mismo terminal.

1. Generamos el ejecutable asociado al archivo ejemplo.c
   `$ gcc exemple.c -o exemple`
2. Generamos una librería dinámica asociada al archivo `malloc_dummy.c`
   `$ gcc -O -shared -fPIC malloc_dummy.c -o malloc_dummy.so`
3. Indicamos, a través de una variable de entorno, que es necesario cargar esta librería dinámica antes que cualquier otra librería
   `export LD_PRELOAD=$PWD/malloc_dummy.so`
   Para que esta instrucción funcione correctamente asegúrese de que el directorio donde está no contiene espacios.
4. Por último ejecutamos nuestra aplicación de forma habitual
   `$ ./exemple`

Al ejecutar de esta forma se utilizará nuestra implementación de `malloc`. Se recomienda probar a ejecutar otras aplicaciones sencillas como los comandos `ls` o `cp`. Todas utilizarán nuestra implementación del `malloc`. Hay que tener en cuenta también que otras aplicaciones habituales pueden no funcionar.

En nuestra implementación de `malloc` no se libera de forma explícita la memoria dinámica. En salir del proceso el sistema operativo se encargará de liberar esa memoria dinámica. En todo caso, para tener un código limpio es necesario liberar la memoria dinámica cuando el proceso no la necesita para asegurar que la aplicación sólo utiliza la memoria dinámica que le hace falta.

3 UNA VERSIÓN DE MALLOC MÁS ADECUADA

Se presenta a continuación una versión de `malloc` más adecuada. En particular, un `malloc` donde se libere memoria reservada utilizando un free.

3.1 ASOCIAR UNA ESTRUCTURA A CADA BLOQUE

Para implementarlo se asocia, para cada bloque reservado con `malloc`, una estructura con la información sobre el bloque reservado, como se ve a continuación.

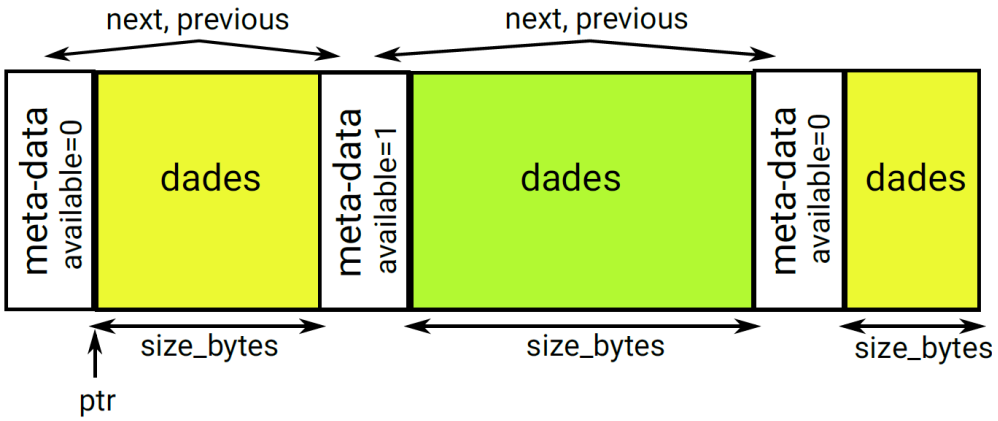

***Figura 1**. Cada blog de datos tiene asociado unos meta-datos que contienen información asociado al bloque.*

Una forma de hacerlo es guardar al principio de cada bloque de memoria información, meta-datos, asociado al bloque. Aquí tenemos la estructura a utilizar, archivo struct.h.

```
#define SIZE_META_DATA  sizeof(struct m_meta_data)
typedef struct m_meta_data *p_meta_data;

/* This structure has a size multiple of 8 bytes */

struct m_meta_data {
    size_t  size_bytes;
    int     available;
    int     magic;
    p_meta_data next;
    p_meta_data previous;
};
```

Dentro del `struct` se definen los siguientes atributos: `size_bytes`, `available`, `magic`, `next` y `previous`.

- `size_bytes` representa el tamaño del bloque solicitado por el usuario, en bytes.
- `available` indica si el bloque está disponible o no. Si no está disponible se está utilizando en ese momento para almacenar cosas. Si está disponible, es indicación de que se ha liberado para que se pueda utilizar.
- El atributo `magic` es un "valor mágico" que se asigna a los metadatos y que podrá utilizar para asegurar que todo funciona correctamente.
- Un apuntador la `next` y `previous` estructura de meta-datos.

3.2 FIRST FIT-MALLOC

Todo el código que se muestra a continuación se encuentra en el código `malloc_first_fit.c`. Nuestra nueva función `malloc` es ésta.

```c
p_meta_data first_element = NULL;
p_meta_data last_element  = NULL;

#define ALIGN8(x) (((((x)-1)>>3)<<3)+8)
#define MAGIC     0x12345678

void *malloc(size_t size_bytes)
{
    void *p;
    p_meta_data meta_data;

    if (size_bytes <= 0) {
        return NULL;
    }

    // We allocate a size of bytes multiple of 8
    size_bytes = ALIGN8(size_bytes);
    fprintf(stderr, "Malloc %zu bytes\n", size_bytes);

    meta_data = search_available_space(size_bytes);

    if (meta_data) { // free block found
      meta_data->available = 0;
    } else {     // no free block found
      meta_data = request_space(size_bytes);
      if (!meta_data)
        return (NULL);

      meta_data->previous = last_element;
      last_element = meta_data;

      if (first_element == NULL) // Is this the first element ?
        first_element = meta_data;
    }

    p = (void *) meta_data;

    // We return the user a pointer to the space
    // that can be used to store data

    return (p + SIZE_META_DATA);
}
```

La variable `first_element` apunta al primer elemento de la lista de bloques reservado. La variable `last_element` apunta al último elemento de la lista. Observar que el valor de `size_bytes` se alinea con un múltiplo de 8 bytes. Esto se debe a que el puntero devuelto por `malloc` debe estar alineado con 8 bytes ya que los procesadores de hoy en día utilizan instrucciones (como las SSE) que requieren este alineamiento.

La función `malloc` utiliza las funciones `search_available_space` y `request_space` para gestionar la memoria. La función `request_space` es la que hace la llamada a sistema para pedir espacio en el sistema operativo mediante la llamada a sistema `sbrk`. Éste es el código:

```c
p_meta_data request_space(size_t size_bytes)
{
    p_meta_data meta_data;

    meta_data = (void *) sbrk(0);

    if (sbrk(SIZE_META_DATA + size_bytes) == (void *) -1)
        return (NULL);

    meta_data->size_bytes = size_bytes;
    meta_data->available = 0;
    meta_data->magic = MAGIC;
    meta_data->next = NULL;
    meta_data->previous = NULL;

    return meta_data;
}
```

La función `free`, que se deberá implementar en esta práctica, es la que deberá liberar el bloque de datos correspondiente (ver sección 4). Para ello basta con poner el valor del atributo `available` a 1. De esta forma se indica que el bloque es libre para futuros `malloc`. A la hora de hacer un `malloc` comprueba primero, mediante la función `search_available_space`, si hay un bloque disponible lo suficiente grande. Si es así, se devuelve al usuario este bloque y se evita hacer un llamamiento a sistema. Éste es el código correspondiente:

```
p_meta_data search_available_space(size_t size_bytes) {
    p_meta_data current = first_element;

    while (current && !(current->available && current->size_bytes >= size_bytes))
        current = current->next;

    return current;
}
```

Analizar bien el código de la función `malloc` que se acaba de definir: en caso de que se encuentre un bloque libre suficientemente grande, se indicará que ya no está disponible. En caso de que no se encuentre ningún blog libre suficientemente grande, se pedirá nuevo espacio con la llamada a sistema `sbrk`. Observar que la función `malloc` devuelve un puntero al espacio de memoria que el usuario puede utilizar (el bloque de datos, ver Figura 1). Los meta-datos se encuentran en memoria, "justo debajo", pero el usuario no debe encargarse de manipularlas.

4 TRABAJO A REALIZAR

Todas las funciones comentadas en la sección anterior se encuentran en el archivo `malloc_first_fit.c`. Comprueba que el código compila y ejecútalo con `exemple.c`. A continuación se propone añadir funciones adicionales a las funciones que ya tiene para que pueda funcionar con otras aplicaciones. Esto implica la realización de los puntos 1 a 5.

1. Implementación de la función `free(void ppt)` que utilice la estructura propuesta. Pare implementar esta función hay que tener en cuenta que se pasa como parámetro a la función `free` un puntero en los datos (la variable `ptr` del dibujo), pero la estructura se encuentra "justo debajo". Básicamente la función `free` debe poner el atributo `available` a 1. Para asegurar que la función hace el trabajo correctamente, compruebe que el atributo `magic` tiene el valor que debe tener (en caso contrario, imprima un mensaje de error). Hay que tener en cuenta que se puede llamar a `free` con un apuntador en `NULL`. En este caso debe ignorarse la llamada. Imprime por pantalla el valor de `size_bytes` que se libera.
2. Una vez implementada la función `free` asegurarse de que funciona correctamente. Pruebe la su implementación utilizando el código `exemple.c` que se muestra al inicio de la práctica.
3. Implementar la función `void *calloc(size_t nelem, size_t elsize)`. La función `calloc` permite reservar varios elementos de memoria, en concreto `nelem` elementos de tamaño `elsize` bytes, y los deja inicializados a cero. Se aconseja utilizar la función `memset` para inicializar el bloque de datos a cero. La función devuelve un puntero a la memoria reservada.
4. Implementar la función `void realloc(void ptr, size_t size_bytes)`. La función `realloc` reajusta el tamaño de un bloque de memoria obtenido con `malloc` a un nuevo tamaño. Se propone que la implementación de la función `realloc` sea la siguiente: a) si le pasamos un puntero `NULL` en `ptr`, se supone que la función `realloc` actúa como un `malloc` normal y corriente. b) si le pasamos a `ptr` un apuntador que hemos creado con nuestro `malloc` y el tamaño que pedimos es suficiente con el bloque

que ya tiene reservado, no hace falta hacer nada, lo devolvemos el puntero tal cual. c) en caso contrario, deberemos reservar un nuevo bloque con más espacio y copiar los datos del antiguo bloque en este nuevo. Por eso se puede usar la función `memcpy` para copiar el contenido de un bloque en otro.

5. Pruebe ahora de nuevo la implementación del `malloc` que tiene. Asegúrese de que la librería funciona con aplicaciones como el `grep` o el `find`. Tener en cuenta que será necesario ejecutar estas aplicaciones desde el terminal donde haya definido el `LD_PRELOAD`.

Se propone realizar una segunda versión de la implementación del `malloc`, `malloc_best_fit_first_fit.c`.

6. Primero de todo, se debe modificar el código para que se haga un *best fit* en lugar de un *first fit*. Es a decir que busque el bloque de tamaño más adecuado.
7. Modifique el código del `free` de tal forma que cuando liberamos un bloque pueda juntar varios bloques contiguos si están vacíos. Aproveche el hecho de que dispone de los atributos `next` y `previous` para evitar tener que recorrer toda la lista de bloques.
8. Asegúrese de que la librería funciona con aplicaciones como el `grep` y el `find`.

De forma opcional, se pide modificar el código del `malloc` y de `realloc` de tal forma que cuando reutilicemos bloques, éstos se puedan dividir en el tamaño necesario.

5 ENTREGA Y EVALUACIÓN

La entrega de esta práctica consiste en dos partes: **el código fuente** (80% de puntuación) y el **informe** (20% de puntuación).

Se debe entregar los siguientes directorios. Los puntos 1 y 2 permiten obtener una calificación máxima de 10 en la práctica, mientras que la parte opcional permite añadir un punto adicional (calificación máxima de 11).

1. Una implementación del `malloc` en el que esté el `free, calloc` y `realloc` utilizando el *first fit*. Incluya también un script que compile y ejecute el ejemplo utilizando la librería `malloc` con la variable de entorno `LD_PRELOAD`. Asegúrese de que la librería funciona con la aplicación `find` o `grep` (a la hora de revisar la práctica los ejecutaremos sin parámetros). El fichero C a entregar debe tener el nombre **`malloc_first_fit.c`**.
2. Una implementación del `malloc` en el que esté el `free, calloc` y `realloc` haciendo y que implemente los puntos 6 y 7 especificados en esta práctica. Incluya también un script que compile los archivos y ejecute el ejemplo utilizando la librería `malloc` con la variable de entorno `LD_PRELOAD`. Asegúrese de que la librería funciona con la aplicación `find` o `grep` (a la hora de revisar la práctica los ejecutaremos sin parámetros). El fichero C a entregar debe tener el nombre **`malloc_best_fit_first_fit.c`**
3. En caso de que desee entregar la parte opcional, entregue en un tercer directorio, integrado con la funcionalidad descrita. El fichero C a entregar debe tener el nombre **`malloc_best_fit_first_fit_opcional.c`**

El informe a entregar debe prepararse en **formato PDF o equivalente** (no se admiten formatos como odt, docx, etc.). Una forma adecuada de estructurar el informe para esta práctica sería definir tres secciones:

1. **Introducción**: se debe describir brevemente el problema que se ha solucionado (es decir, un resumen de lo que propone esta práctica).
2. **Implementación y pruebas realizadas**: por un lado, de pide describir cómo se ha estructurado el código, enumerando las distintas funciones y proporcionando una breve descripción del cometido de cada una de ellas. También se puede destacar la definición

de alguna variable que se considere importante. Por otro lado, también se han de mostrar las pruebas que se han realizado para asegurar el buen funcionamiento del código.

3. **Conclusiones**: se debe hacer una breve reflexión sobre lo aprendido. De manera opcional, también se puede usar esta sección para aportar una reflexión personal sobre la práctica, incluyendo sugerencias sobre aspectos que se podían haber incluido a fin de hacerla más provechosa.

En el caso de que se desee incluir capturas de pantalla en lugar de incluir los resultados de los experimentos en formato texto, es necesario asegurarse de que el texto de la captura se puede leer bien (es decir, que tenga un tamaño similar al resto del texto del documento) y que todas las capturas sean uniformes (es decir, que todas las capturas tengan el mismo tamaño de texto).

El documento debe tener una longitud máxima de 4 páginas (sin incluir índice ni la portada). El documento se evaluará con los siguientes pesos: pruebas realizadas y comentarios asociados, un 60 %; escritura sin faltas de ortografía y/o expresión, un 20 %; paginación del documento hecha de forma limpia y uniforme, 20 %.

Cada grupo debe subir **un único archivo ZIP** a la tarea asociada a esta práctica en el campus virtual con el nombre **P4_GrupoXX_nombre1_apellido1_nombre2_apellido2.zip**, y debe incluir todos los archivos descritos anteriormente.