UNC Universidad Nacional de Córdoba

FCEFyN Facultad de Ciencias Exactas, Fisicas y Naturales

TRABAJO PRÁCTICO Nº 4:

“memory management”

Cátedra: Sistemas Operativos 1

Año: 2019

Integrantes:

- Ferrero Alejandro (Matrícula n° 40054394)
- Pichetti Augusto (Matrícula n° 41018392)

Carrera: Ingeniería en Computación.

Profesores:

- Martínez Pablo
- Alonso Martín

Introducción

En el presente Trabajo Práctico se desarrollan 2 algortimos de asignación dinámica de memoria: first-fit y best-fit. En ambos casos se hace uso de las funciones sbrk() y brk() para modificar la posición del break o límite del heap y se diseñaron estructuras que contienen información de un bloque de memoria particular y punteros al bloque anterior y siguiente de forma que la memoria se estructure como una lista doblemente enlazada.

Desarrollo del proyecto

First-Fit

En esta parte se implementaron las funciones malloc(), realloc(), calloc() y free(). A su vez, presenta otra funciones auxiliares para el manejo del heap, búsqueda y manejo de bloques, etc.

El metadato o encabezado de cada bloque de memoria asignado se implementó mediante la siguiente estructura:

```
struct s_block {
    size_t size;
    struct s_block *next;
    struct s_block *prev;
    int free;
    void *ptr;
    /* A pointer to the allocated block */
    char data[1];
};
```

Donde size es el tamaño del bloque correspondiente al metadato (si contar el espacio ocupado por el mismo). Next y prev son punteros a los metadatos siguiente y previo respectivamente. Free es un flag que indica si el bloque está o no asignado. Data es un puntero al bloque asignado y ptr es otro puntero que apunta la compo data, el cual se usa para verificar en la función free() que un puntero haya sido asignado por malloc() y para saber si es un bloque válido.

Funciones implementadas:

- **void *malloc1(size_t size)**: reserva un bloque de memoria y devuelve un puntero void al inicio de la misma. El parámetro size especifica el número de bytes a reservar.
- **void *realloc1(void *p, size_t size):** redimensiona el espacio asignado de forma dinámica anteriormente a un puntero. Donde p es el puntero a redimensionar, y size el nuevo tamaño, en bytes, que tendrá. Si el puntero que se le pasa tiene el valor nulo, esta función actúa como malloc.
- **void *calloc1(size_t number , size_t size)**: funciona de modo similar a malloc, pero además de reservar memoria, inicializa a 0 la memoria reservada. El parámetro number indica el número de elementos a reservar, y size el tamaño de cada elemento.

- **void free1(void *p)**: sirve para liberar memoria que se asignó dinámicamente. Si el puntero es nulo, free no hace nada. El parámetro ptr es el puntero a la memoria que se desea liberar.
- **#define align4(x) (((((x)-1)>>2)<<2)+4):** macro que se utiliza para alinear los punteros (o bloque de datos) al tamaño de integer (4 bytes, el cual es el mismo tamaño que el de un puntero). Por lo que los bloques serán de múltiplos de 4 bytes.
- **void split_block(t_block b, size_t s):** divide el bloque pasado por argumento para hacer un bloque de datos del tamaño requerido. El bloque b debe existir previamente.
- **t_block extend_heap(t_block last , size_t s):** extiende el heap moviendo el break e inicializando un nuevo bloque. Hace uso de la función sbrk(). Retorna NULL en caso de que sbrk falle.
- **void copy_block(t_block src , t_block dst):** copia los datos desde un bloque origen a un bloque destino.
- **t_block get_block(void *p):** obtiene el bloque (puntero al metadato) del puntero al bloque de memoria pasado como parámetro.
- **int valid_addr(void *p):** verifica que sea una dirección válida para liberar, es decir, que este entre la base y el break.
- **t_block fusion(t_block b):** fusiona el bloque pasado por parámetro con su sucesor.
- **t_block find_block(t_block *last , size_t size):** encuentra un bloque libre lo suficientemente ancho. Comienza en la dirección base del heap, prueba el fragmento actual, si se ajusta a las necesidades, devuelve su dirección, de lo contrario, continúa con el siguiente fragmento hasta que encuentre uno adecuado o el final de la cabeza. Mantiene el último fragmento visitado, por lo que la función malloc() puede extender fácilmente el final del heap si no encuentra un fragmento adecuado.

Best-Fit

Implementa las funciones malloc() y free().

El metadato o encabezado de cada bloque de memoria asignado se implementó mediante la siguiente estructura:

```
struct mab {
    int allocated;    // si el bloque esta ocupado
    size_t size;            // tamaño del bloque (incluyendo header)
    struct mab *prev;   // puntero al bloque previo
    struct mab *next;   // puntero al bloque siguiente
};
typedef struct mab *mabPtr;
```

Donde allocated indica si el bloque correspondiente al metadato está ocupado o no, size, que indica el tamaño del bloque, incluyendo el del encabezado, prev y next son punteros a los encabezados de los bloques previo y siguiente respectivamente.

Se utilizan los punteros head y tail para tener un control de los bloques asignados en memoria, siendo head el primer bloque y tail el último

Funciones implementadas:

- **void *malloc(size_t size):** reserva un bloque de memoria de igual tamaño o aproximado al tamaño solicitado, en caso que no se encuentre un bloque de estas características, se lo agrega al final de la “lista”. Después de asignarlo, se verifica si el tamaño total del bloque en el cual a sido asignado contiene un espacio libre mayor al espacio ocupado, en este caso el bloque se divide en dos.
- **void free(void *ptr):** libera memoria asignada dinámicamente. El puntero ptr es el puntero a la memoria que se desea liberar. En caso que un bloque adyacente al liberado esté también libre, estos se unirán en un bloque de mayor tamaño.