Memoria Dinámica

Introducción

Existen varios motivos por los cuales podríamos solicitar memoria en tiempo de ejecución, uno es porque podríamos necesitar un bloque de memoria de una dimensión que se va a conocer en tiempo de ejecución, otro motivo más rebuscado podría ser desacoplar el ciclo de vida de un dato del ciclo de vida de una función (llamada, ejecución, retorno). Esta flexibilidad también introduce la responsabilidad de gestionar la memoria de forma adecuada para evitar problemas como fugas de memoria.

Similar a lo que sucede con archivos que por cada llamada a `fopen()` hay, en algún lugar no muy lejano del código un `fclose()`, en memoria dinámica siempre que hay una solicitud de asignación de memoria habrá en algún lugar, tal vez tan tarde como justo antes de finalizar el programa, una llamada a `free()` para liberar esa memoria.

De los motivos mencionados anteriormente, el que más nos interesa es el primero y su principal utilidad será la de arreglos dinámicos. Estos arreglos tienen dos vertientes que no son mutuamente excluyentes, los arreglos “del tamaño justo”, es decir que tienen la dimensión adecuada para almacenar todos los datos que se pretenden almacenar y no desperdiciar un sólo byte en memoria y los arreglos que se redimensionan de manera programática, es decir que automáticamente. Antes de insertar un dato o luego de hacerlo se verificará si hay que redimensionar el arreglo y si es así se ampliará su capacidad.

Estructura de la memoria de un proceso

Para comprender la asignación dinámica de memoria, es útil saber cómo se organiza la memoria de un programa en ejecución. Principalmente, encontramos dos regiones importantes: el stack (pila) y el heap (montículo o montón).

- El Stack: Esta región se utiliza para la gestión de las llamadas a funciones y las variables locales. La memoria en el stack se asigna y se libera de forma automática y en un orden LIFO (Last-In, First-Out). Cada vez que se llama a una función, se crea un nuevo "marco" en el stack para sus variables locales. Al finalizar la función, este marco se elimina. La gestión del stack es rápida y eficiente, pero su tamaño suele ser limitado y la vida de las variables está ligada a la de la función.
- El Heap: Esta es la región de memoria donde se realiza la asignación dinámica. A diferencia del stack, la memoria en el heap se gestiona explícitamente por el programador mediante funciones como malloc y free. El heap permite solicitar bloques de memoria en cualquier momento y liberarlos cuando ya no son necesarios. Esto proporciona una gran flexibilidad para gestionar datos cuyo tamaño o tiempo de vida no se conoce en tiempo de compilación. Sin embargo, requiere una gestión cuidadosa para evitar problemas como la fragmentación y las fugas de memoria.

malloc()

La función `malloc()` es la principal en materia de gestión de memoria dinámica, ésta recibe un `size_t` por parámetro (recordar que `size_t` es un alias para `unsigned long long` que, en la mayoría de los compiladores si no es que en todos, es un bloque de 8 bytes que representa un número entero mayor o igual a cero) y retorna un puntero (`void*`) al bloque de memoria solicitado.

Haciendo una breve reseña histórica podemos encontrar sentencias distintas de código. Considérese la siguiente sentencia de código:

```
int* arr1 = (int*)malloc(50 * sizeof(int));
```

En este ejemplo se hace un casteo explícito a puntero a entero porque anterior a la especificación del estándar C99 no existía tal cosa como `void*` y la función `malloc()` retornaba un `char*`, por este motivo era **necesario** castear explícitamente ese puntero al tipo de dato deseado para indicar al compilador cómo interpretar el dato detrás de la dirección retornada. El siguiente ejemplo es igual de válido.

```
int* arr2 = malloc(50 * sizeof(int));
```

Asimismo, malloc() puede ser utilizado para prolongar el ciclo de vida de un dato más allá del lugar donde se declara ese dato.

```
Alumno* aux = malloc(sizeof(Alumno));
```

La dirección de `aux` puede ser devuelta por una función y esa memoria liberarse más tarde.

realloc()

La función `realloc()` es la encargada de gestionar la redimensión de un bloque de memoria, ésta recibe un puntero a un bloque de memoria (previamente solicitado con `malloc()`) y una nueva dimensión que puede ser menor o mayor a la actual. El procedimiento devolverá un puntero al nuevo bloque si se tuvo que reubicar o al mismo bloque si no fue necesario.

Considere el siguiente caso, se tienen 200 bytes reservados mediante `malloc()` y se necesitan 50 más, se realiza una llamada a `realloc()` solicitando 250 bytes y hay espacio libre contiguo, se reserva esa memoria y la dirección base de ese bloque no cambia. Esto es deseable para minimizar la fragmentación de la memoria pero en el caso de que no haya memoria suficiente contigua el procedimiento moverá el el bloque original sin perder ningún dato a una nueva ubicación y retornará la dirección correspondiente.

free()

Por último pero no menos importante `free()` es la función que permite liberar la memoria para su posterior reutilización, ésta recibe un puntero a un bloque de memoria previamente reservado por `malloc()`. Es importante reconocer que pasarle una dirección de memoria que corresponde a la región del stack conduce a comportamiento indefinido que puede corromper los datos en esta región, un fallo de segmentación que cerraría el programa o inconsistencias difíciles de diagnosticar.